Appeared in the Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), ACM Press, Dubrovnik, Croatia, pp. 351-360, September, 2007.

# A Socio-Technical Framework for Supporting Programmers

Yunwen Ye[1,3]
[1]L3D Center
Department of Computer Science
University of Colorado, Boulder, USA
+1-303-492-3547
yunwen@colorado.edu

Yasuhiro Yamamoto[2]
[2]KID Lab
RCAST
University of Tokyo, Japan
+81-3-5452-5286
yxy@kid.rcast.u-tokyo.ac.jp

Kumiyo Nakakoji[2,3]
[3]SRA Key Technology Lab
3-12 Yotsuya, Shinjuku
Tokyo, Japan
+81-3-3357-9011
kumiyo@kid.rcast.u-tokyo.ac.jp

## ABSTRACT
Studies have shown that programmers frequently seek external information during programming, from source code and documents, as well as from other programmers because much of the information remains in the heads of programmers. Programmers therefore often ask other programmers questions to seek information in a timely fashion to carry out their work. This information seeking entails several conflicting factors. From the perspective of the information-seeking programmer, not asking questions degrades productivity. Conversely, asking questions interrupts other programmers and degrades their productivity, and may be frowned upon by peers due to the perceived social inconsideration of the information seeker. From the perspective of the recipients of the question, even though helping is costly, not helping also incurs social costs due to the deviation from social norms. To balance all these factors, this paper proposes the STeP_IN (Socio-Technical Platform for In situ Networking) framework to guide the design of systems that support information seeking during different phases of programming. The framework facilitates access to the information in the heads of other programmers while minimizing the negative impacts on the overall productivity of the team.

## Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *computer-aided software engineering*. H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces - *computer-supported cooperative work, theory and models*. J.4 [**Computer Applications**]: Social and Behavioral Sciences - *sociology*.

## General Terms
Design, Economics, Human Factors

## Keywords
Programming support, socio-technical support, information acquisition and sharing, communication



## 1. INTRODUCTION
Programming involves both individual activities and collaborative activities. As an intensive cognitive activity, programming requires undivided attention, and in general programmers prefer to work in a solitary environment with long periods of uninterrupted time during which they can concentrate [11, 27]. It is this kind of uninterrupted solitary work that gets the code written. However, due to the intricate interdependency of their work caused by the division of labor and the distribution of information required for the creation of software systems, programmers also have to interact with peers for various reasons [16].

Programming is a continual process of seeking information and creating information in representational media. The information created by one programmer affects other programmers whose work depends on it, and who have to seek the necessary information in a timely fashion to carry out their work. In programming, code and documents are used as media for representation, and they are the major resources for programmers to seek necessary information. However, programmers do not always articulate everything in code and documents, and much of the information remains in their heads [19]. Programmers therefore often have to interrupt—or be interrupted by—other programmers through various communication channels to accomplish their individual programming tasks effectively.

This kind of information seeking is not for the general purpose of learning or creating awareness, in which information is not immediately coupled with the task at hand. Rather, it is a clearly purposed act that serves the goal of accomplishing an individual programmer's task at hand; it arises on an as-needed basis and requires quick resolution. A programmer who is unable to obtain such information timely cannot carry out his or her programming effectively, thus lowering that programmer's productivity, which in turn lowers the productivity of the project team.

Asking peers for information, however, incurs costs that also have negative impacts on the productivity of the team. Information seeking interrupts the preferred solitary work of the programmers who are asked. The cost of interruption is not only the time and attention spent to help the information-seeking programmer; it also is the disruption of flow and continuity of the ongoing work, which reduces the productivity of the helper [29]. Although helping the information-seeking programmer is costly, saying "no" to peers would also incur considerable social cost.

A trade-off, therefore, exists between the frequency and the timing of programmer interactions for the purpose of information seeking that arises during programming practices. Computational

tools that support a programmer's seeking of external information to improve his or her productivity have to be carefully balanced with incurred cost on the overall productivity of the team as a whole.

We approach the above issue from a socio-technical perspective. In addition to providing potentially relevant information for the current context, we argue that programming environments also need the integrated support of helping programmers in asking questions to acquire information held in the heads of peer programmers. We propose the STeP_IN (Socio-Technical Platform for In situ Networking) framework to provide unified support for acquiring external information in code and documents and with peers. This general framework is intended to provide guidelines for developing programming environments that support programmers in acquiring external information during different phases of their programming practices. To examine the feasibility of the STeP_IN framework, we have instantiated the framework and developed a socio-technical system of supporting Java programmers to learn to use class libraries.

## 2. SOCIAL FACTORS OF PEER SUPPORT

A number of researchers have already recognized the need to use the expertise of peer programmers. Berlin has found that expert programmers are experts not only because they have more expertise but also because they use other programmers' expertise more frequently [4]. Several systems that help programmers find experts, notably Expertise Recommender [21] and Expertise Browser [23], have been proposed in the past years.

Finding experts, however, does not necessarily lead to the acquisition of the information being sought. As information resources, programmers differ from 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 [17]." The willingness of a programmer to engage in helping other programmers to solve their tasks depends on a set of perceived social factors.

Note that the social factors considered here differ from those of existing studies that focus on social aspects of software development in such contexts as how developers and users work together in designing a computer system [32], or how developers coordinate their work in geographically distributed projects [16].

The following social factors need to be taken into account when programmers want to use peers as information resources for their programming tasks.

*(1) Programmers have to take the risk of looking "ignorant" among their peers because asking a question implies they are missing some knowledge*. Studies show that askers demonstrate different asking behaviors, depending on whether they are in public, in private, communicating with a stranger, or communicating with a friend due to the different levels of perceived psychological safety in admitting a lack of knowledge [7].

(2) *Programmers feel different levels of difficulty and easiness, depending on to whom they ask and through what communication channels*. It is easy for programmers to ask peers for information through face-to-face communication because they tend to have a good sense of what the person is doing at the moment of asking and feel socially comfortable to initiate contact with those they know well. People feel relatively comfortable posting a question to a mailing list if they are familiar with some members of the mailing list. It is usually easier to compose a question in a personal email communication than to post a question to a Wiki or other type of publicly accessible media.

(3) *Programmers may immediately get necessary information or may never get any useful information, depending on to whom they ask and how they ask*. Rhetorical strategies, linguistic complexity, and word choice of the question all influence the likelihood of others responding [18]. Making a personal appeal in the question results in a better and faster response than making a non-personal appeal [7].

(4) *Programmers may feel interrupted when being asked for information by their peers*. Answering, or providing help, consumes the time and attention of the helpers and interrupts their primary task. An interruption is regarded as an unexpected encounter initiated by another person, which disturbs "the flow and continuity of an individual's work and brings that work to a temporary halt to the one who is interrupted" [29].

(5) *Programmers may not be willing to respond to a question, depending on who is asking what*. Responding to a question requires programmers to spend precious time. Deciding whether and how to help an asker depends on their perceived social relationship both with the asker and with the social environment at large. The theory of social capital provides an analytic framework to understand this decision-making process [13]. Social capital is the "sum of the actual and potential resources embedded within, available through, and derived from the network of relationships possessed by an individual or social unit" [24]. Social capital is regarded as important as financial capital and intellectual capital for an individual as well as a social organization because it would promote cooperation and reduce transaction costs [15].

(6) *Programmers may respond to a question not because they want to answer it, but because they do not want to ignore it*. Even though helping is costly, taking no action also incurs a social cost. Saying "no" untactfully to an asker deteriorates the expert's relation with the asker and negatively affects the expert's social reputation among other peers because such behavior deviates from social norms [26].

## 3. THE STEP_IN FRAMEWORK

We have developed a conceptual framework called STeP_IN for the design and development of socio-technical environments that support programmers to seamlessly access external information, which includes source code, documents, and peers as information resources that are necessary for their programming tasks. We envision that this framework will serve as a guiding substrate for other researchers as well as ourselves in developing application systems that support different information needs in different phases of programming, such as those for program maintenance and those for learning a component library.

The STeP_IN framework views peer programmers as information resources equally important as code, documents, and various kinds of information repositories (e.g., reuse repositories, discussion archives) that programmers might use during programming practices. It takes into consideration the social factors of treating peer programmers as information resources, and employs mechanisms that ensure communications between programmers that are not disruptive to the overall productivity as well as social atmosphere of the team as a whole.

Many types of communication needs exist in programming, such as those of informing programmers of the status of the project, those of brainstorming for design ideas, and those for consensus building. STeP_IN aims to support the kind of workflow-based communication that facilitates information seeking and sharing during the ongoing activities of an individual member. This type of communication is characterized by such features as the following: it arises on an as-needed rather than scheduled basis; it is usually for problem-solving and assistance-giving in nature; it occurs as a sequence of highly focused interactions in a short period of time; and it involves a relatively small group of participants. This type of ad hoc and in situ communication that supports programmers to carry out their individual tasks has been shown to take up to 41% of a programmer's time [28].

## 3.1 Design Principles

This section outlines the essential principles that underlie the design of the STeP_IN framework.

*Principle 1: Information seeking should be treated as an in situ and highly individualized act.* A programmer's needs for external information arise when he or she is dealing with a specific programming task in a programming environment. The acquisition of the necessary information for a programmer therefore has to be made in a timely fashion so that he or she can carry out the current task more effectively and productively in a fluid manner [33]. The external information also needs to be presented to the information-seeking programmer in a way that is shaped to the context of the particular problem and the unique structure of the programmer's background knowledge.

*Principle 2: Whenever possible, programmers should minimize the times to resort to peers as information resources.* Resorting to code and documents is inexpensive in the sense that the use of such information is merely at the bid of the information-seeking programmer. The major challenge of supporting the easy acquisition of information existing in code and documents is mostly technical: to design better information retrieval mechanisms and interfaces. In contrast, resorting to peers as information resources comes with a much higher social cost and is highly constrained because it involves not only the information-seeking programmer but also those who are asked to provide information.

*Principle 3: When a programmer seeks information from peer programmers, he or she should be able get the necessary information in a timely fashion with minimum effort.* When a programmer has to resort to peers as information resources, he or she should be able to ask those who have the pertinent information, regardless of whether the programmer knows who the experts on the particular topic are. The information that the information-seeking programmer gets should be of high quality, arrive timely, and help the programmer to solve the current problem.

*Principle 4: The interruption caused by being asked for help by other programmers should be reduced as much as possible.* When a programmer is approached to provide information for the benefit of another programmer, that programmer is distracted from his or her own programming task. The cost of interruption incurred on each information-providing programmer should be minimized through the use of appropriate communication tools. More important, from the perspective of the collective productivity of the team, those programmers who do not provide such information should not be interrupted.

*Principle 5: Programmers should not be forced into sharing information.* Helping peers by providing information should not be treated as an isolated one-shot act in the process of software development; it has to recur as new needs emerge. The success of one act of providing information should not come at the cost of the information provider's reluctance to engage in future acts of providing information. Because the willingness of information providers is the critical factor that affects the long-term success of the project, it is important to grant information providers the full freedom of deciding how they want to engage in helping others. They should not be forced into helping just for the fear of causing unnecessary disruptions to the social cohesion and norms of the project team, which is unlikely to be sustainable.

## 3.2 Software Project as a Socio-Technical Information Space

To create programming environments that adhere to the above principles, the STeP_IN framework conceptualizes a software project as a *socio-technical information space*, which consists of three interrelated elements: *code*, *documents* and *programmers*. The term "document" here refers to both the traditional design documents (e.g., specifications, test plans) and such records as configuration management logs, bug reports, and email archives that are accumulated during the development of systems. In this socio-technical information space, information is embodied in both its constituting elements and the relations of its elements.

Six categories of relations among constituting elements exist in the socio-technical information space:

- *code-code*: A piece of code can be related to other pieces of code in many different ways. For example, they are related through their control flow or data flow, which defines the order of execution; they are related if they both are part of the implementation of a domain concept; and they are deemed potentially related if they have been frequently modified at the same time by many developers [37].
- *code-document*: A piece of code is related to a document if it implements some of the concepts contained in that document based on traceability analysis [3]; or if the document, such as a Concurrent Versions System (CVS) log, explains the development history of the code [8]; or if the code is accompanied by a reference documentation, such as in the case of reusable library code.
- *document-document*: A document is related to another document if their contents are related to each other. For example, a bug report is related to emails that discuss the bug.
- *programmer-code*: A programmer is related to a piece of code if he or she has created, modified, or used it. This relation indicates that the programmer might have expertise about the code or retain certain information about the code that has not been articulated in the code and its associated documents.
- *programmer-document*: A programmer is related to a document if he or she participated in the discussion, creation, or modification of the document. This relation indicates that the programmer might have contextual and tacit knowledge about the document.
- *programmer-programmer*: The relation between programmers captures the history of their social interactions, including who has helped whom, who prefers to work with whom, and who has sent emails to whom.

These six categories of relations are defined differently when the STeP_IN framework is instantiated for different purposes.

We view that the external information that a programmer needs to acquire for his or her task lies in nodes of the socio-technical information space and the structural configuration of the nodes. To obtain such information, the programmer thus has to be able to traverse the relational links among such nodes because information flows along those links. Those relational links are not immediately discernable to programmers when the information space is huge and complex. The STeP_IN framework tries to provide a programmer with an integrated socio-technical solution to the in situ and individualized acquisition of external information, regardless of whether its source is code, documents, or peer programmers, by helping the programmer to traverse the relational links while taking into account social considerations (Figure 1).

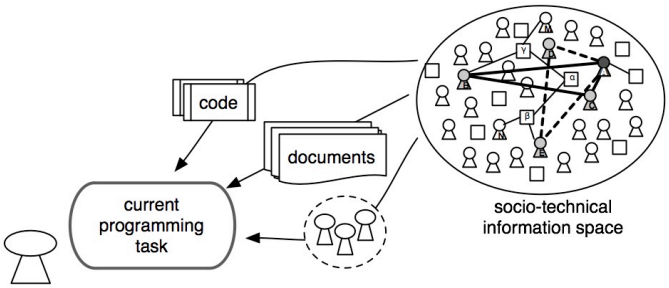


**Figure 1: Socio-Technical Support for Programmers**

## 3.3 STeP_IN Features

The following features of the STeP_IN framework ensure that an instantiated programming support environment adheres to the design principles outlined in Section 3.1.

### *3.3.1 Integration with Working Environment*

Support for information acquisition needs to be tightly integrated with a programmer's working environment for two reasons. First, the support needs to be tuned to the existing working process of the programmer because he or she needs to acquire new information for the purpose of applying the new information to the current task. When information acquisition support exists as a stand-alone system, it increases the cognitive cost of information access and use [5], such as requiring conscientious mental switch of different workspaces, losing short-term memory, and needing to reconstruct working context.

Second, integration with the working environment provides the context of the problem with which a programmer is dealing, which can be utilized by the information acquisition support system to customize its support to the specific context and the background knowledge of the programmer [8, 35].

### *3.3.2 Presentation of Task-Relevant Information*

When a programmer lacks information to accomplish his or her task under the pressure of productivity, the programmer would generally be interested only in the information that helps to finish the task. The challenges for a programmer to acquire task-relevant external information are [12]:

(1) The programmer might not be aware that task-relevant information even exists; and/or
(2) The programmer has to be able to start the search with a well-formed query to explore the information space.

The STeP_IN framework incorporates both information delivery and information access mechanisms to help programmers start the information acquisition process. The information delivery approach [34] analyzes the context in which the programmer is currently working, and proactively prompts programmers to task-relevant information whose existence the programmer might not know. The information access mechanism provides a query interface that allows programmers to locate the most relevant information when they know what they are looking for. This phase of support is based on the relation that directly links the code and document to the programmer's current task, and returns first-cut search results (by either delivery or access) that are immediately relevant to that task.

The relation utilized to return these first-cut search results is different when the STeP_IN framework is instantiated to support different programming tasks. For example, if we are to develop a STeP_IN-based tool that supports library reuse, the first-cut results would utilize the conceptual similarity between a programmer's coding task and reusable components in the library [35]. If we are to develop a STeP_IN-based tool that supports the maintenance of code, the first-cut search results could be based on traceability links between code and design documents [3] or mined implementation links between code and design issues [8].

### *3.3.3 Incremental Access to Contextual Information*

If the immediately relevant information is not enough for the programmer to accomplish his or her task, the programmer should then be able to traverse the relations between the first-cut search results and other code and documents in the socio-technical information space to find more information that provides the context. Such contextual information can help the programmer to better interpret and understand the information pieces in the first-cut search results. For example, the programmer can find example programs that use a library code for reuse support, or the programmer can find the development history of a piece of code for maintenance support.

### *3.3.4 Identification of Peers to Ask for Information*

If a programmer is still not able to accomplish the current task with the information embodied in code and documents that he or she could find, the programmer then should be able to ask for help from peers who might have the information.

The STeP_IN framework takes into account *what* information is sought by *whom* in identifying peers for the information-seeking programmer to contact. The information pertaining to the former is called the *technical profile* and the information pertaining to the latter is called the *social profile*. The framework does not inform the information-seeking programmer who the identified peers are. Instead, it provides a communication channel for the information-seeking programmer through which he or she can pose a request for more information about a piece of code or document without the need to know the identity of the experts. The question posed by the information-seeking programmer is automatically routed to a group of peer experts. Section 3.3.5 describes details of such a communication channel.

The most important element here is to identify peers who are willing to provide high-quality information in a timely manner to the information-seeking programmer. At the same time, such a request should not put any undue pressure on peers to provide information if they are not in the appropriate condition, or willing, to do so for whatever reasons.

First of all, the peer needs to have expertise on what is sought. Additionally, as discussed in Section 2, such expertise-holding programmer has to be willing to engage in providing information with the information-seeking programmer. This willingness varies according to the perceived social relationship with the particular information seeker, as well as the perception of such factors of social capital as obligations, expectations, and norms of general reciprocity in the social group to which both programmers belong.

#### *3.3.4.1 Technical Profiling: Considering What Information Is Sought*

The STeP_IN framework identifies peers that have the *programmer-code* and *programmer-document* links in the socio-technical information space to a piece of code or document in which the information-seeking programmer is currently interested. This process is similar to that of the Expertise Browser system [23,] which finds programmers who have previously worked on the code based on CVS logs.

A *technical profile* of a programmer indicates his or her expertise. STeP_IN may use different kinds of links between programmer and code as well as programmer and document to develop the technical profile of a programmer. The technical profile of a programmer may first be initialized by mining the historical data of software projects and the programmer's past work. It is also important that the technical profile be editable by each individual programmer to choose to answer only those questions in which they are interested and to reduce the number of interruptions for questions that they do not like and probably would not answer. People are generally more interested in answering questions for which they think they have exclusive expertise [6].

#### *3.3.4.2 Social Profiling: Considering Who Is Seeking Information*

The STeP_IN framework identifies peers based on the programmer-programmer relations that exist in the socio-technical information space. It models a programmer's social relations with other programmers in the *social profile* of that programmer. The social profile of a programmer has three major components: *Inter-Personal Obligation*, *Total Social Obligation*, and *Inter-Personal Preference*. The first two components may be derived from the programmer's previous interactions with the particular information-seeking programmer as well as with other peers in the whole group. The last component should be editable by individual programmers.

(a) *Inter-Personal Obligation* denotes and models the inter-personal social obligation that programmer X owes to programmer Y. It denotes whether programmer X has certain obligations to help information-seeking programmer Y because X has been helped more by Y.

(b) *Total-Social Obligation* denotes and models the social obligation that programmer X owes to the whole group. It denotes whether X has certain obligations to help any other programmers because X has often been the recipient of help in the past.

(c) *Inter-Personal Preference* denotes a programmer's individual preference of collaborating with each of other programmers. Note that this relationship is not reflexive because each party's perception of their relation is quite different, and not always mutual.

Among many social factors, of particular interest regarding a programmer's willingness to engage in helping another peer are the expectations that other peers have of the individual programmer and the obligations that the individual programmer feels toward others. The expectations and obligations result from previous social interactions among programmers.

Social relations are nuanced and affected by the subjective perception of each person. It is quite natural that a person may prefer to help, or not help, another person no matter what social obligations he or she has. A situation in which a programmer is forced into helping another programmer whom he or she does not like to help just because he or she should do so, often results in unsuccessful information sharing. Previous research has reported that, when some experts are approached by people with whom they do not like to work, they often quickly craft an impressive-sounding, but not helpful, answer as a social defense to meet the minimum demand of acceptable social behaviors and at the same time to keep the information seekers from consuming too much of their time [7]. Such behaviors just waste the time and attention of both the information seeker and the information provider.

#### *3.3.4.3 Concealed Identities*

By taking into account both the technical and social profiles of peer programmers, the framework becomes able to identify a group of peers who both have the expertise and are highly likely to offer help to answer the question posted by the information-seeking programmer. However, the STeP_IN framework does not show the identified group of peer programmers to the information-seeking programmer. This design decision ensures that the information providers, rather than the information seeker, have control over the information-sharing process.

The benefits of the information-sharing process are asymmetrical in that the information seeker enjoys far more benefits than information providers; yet, if the information seeker knows directly who the experts are, he or she decides, at his or her own convenience, when and who to interrupt. Many organizations that have published expert lists on their intranet have soon received many requests from those listed experts asking to be de-listed due to the overwhelming interruptions to their work.

### *3.3.5 Creation of a Socially-Aware Communication Channel*

When a group of peer programmers who are likely to provide high-quality information are identified, the framework provides a communication channel through which the question posed by the information-seeking programmer is automatically routed to the identified peers. The STeP_IN framework currently offers an ephemeral mailing list for this purpose.

The question is sent to each member of the ephemeral mailing list through emails. If a member of the ephemeral mailing list replies to the question, his or her answer is sent to all members on the list. The information exchanged is archived and stored in the socio-technical information space for the benefit of those programmers who are not involved. This ephemeral mailing list is different from the traditional mailing list in that it is dynamically created each time a question is posted and will disappear when the information-sharing act is finished. The recipients are not determined by their own subscriptions, but are selected based on their social relationships with the information seeker and their technical expertise on the topic.

The latter point makes the ephemeral mailing list similar to direct emails because the members are intentionally targeted recipients who have already established social ties with the sender. However,

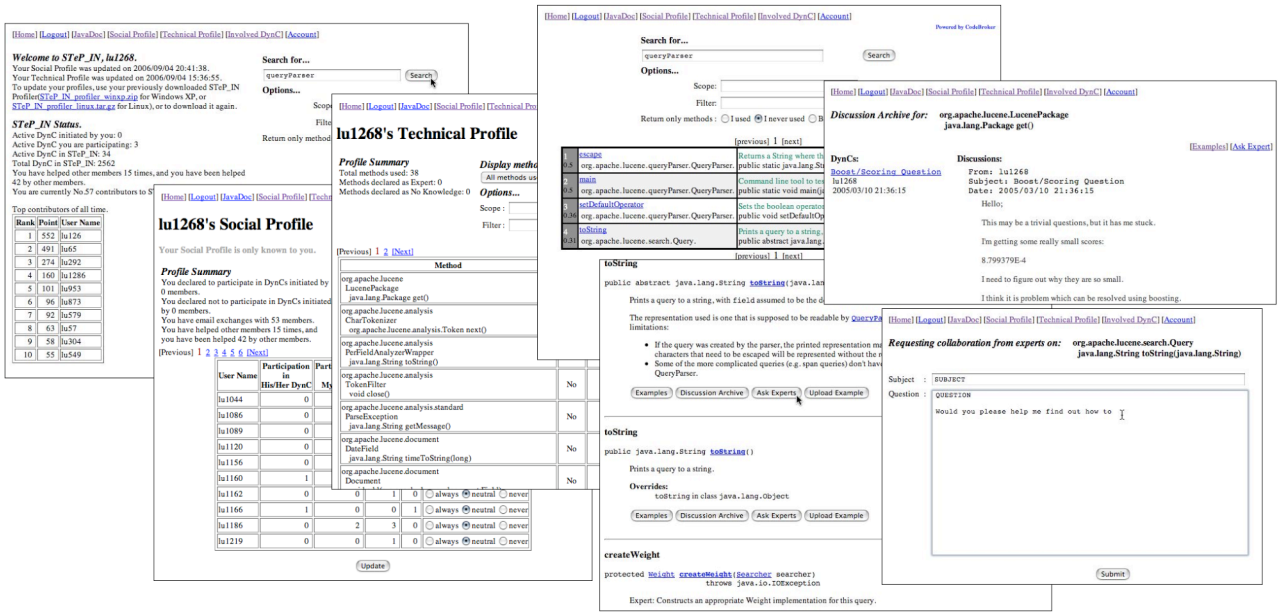

**Figure 2: SIJ: Instantiated from the STeP_IN Framework**

the list differs from direct emails in that the members of the dynamically created ephemeral mailing list are not made public. The information-seeking programmer does not know who has received his or her question. Only the receiver of the question knows from whom he or she had received a question, but does not know who else has received the same question. Because of this anonymity, the recipient of the question has the freedom to decide his or her engagement in the information-sharing act, and on what terms, without the worry of being cornered into certain actions just to meet the demands of acceptable social behaviors but at the cost of his or her own productivity (and also the team productivity).

If the recipient does not answer, for whatever reasons, nobody will know it; therefore refusing to help is not socially unacceptable, similar to "hiding out to get some work done" [11]. However, if the recipient answers the question, his or her identity is revealed to all members of the ephemeral mailing list. This asymmetrical information disclosure is meant to reinforce positive social behaviors without forcing others into collaboration that comes at the cost of either degrading the productivity of expert programmers or damaging their willingness to engage in information providing in the future.

Meantime, this process also relieves some of the social burdens of the information-seeking programmer in deciding to ask a question. Knowing that other programmers have the freedom to not respond, the information-seeking programmer does not need to consider whether his or her question would create a burden for the recipient.

# 4. APPLYING THE FRAMEWORK

To assess the feasibility of the STeP_IN framework, we have instantiated the framework and developed the SIJ (STeP_IN for Java) system to help Java programmers acquire external information about Java class libraries.

## 4.1 The SIJ System

By incorporating social support, SIJ (Figure 2) extends the CodeBroker system [35] that we previously developed to support Java programmers to learn to use class libraries from documents, examples, and other peers in a seamless way. The vast amount of class libraries (for example, the Java SDK alone has 3279 classes) makes it almost impossible for programmers to know all of them. The not-known-yet part of the class libraries thus becomes the external information that programmers have to utilize to develop software systems efficiently.

### *4.1.1 Finding Task-Relevant Library Methods*

The SIJ system first makes programmers aware of unknown class libraries by proactively delivering *task-relevant* and *personalized* library methods that can potentially be applied in the current programming task but are not yet known to the programmers (see [35] for details). The SIJ system is seamlessly integrated with the programming environment Emacs. It has an interface agent that runs as a background process in Emacs. The interface agent analyzes continuously the partially written program in Emacs, infers what kind of class libraries can be applied in the current task, and displays automatically, at the bottom of the editing space, the names and short descriptions of those task-relevant library methods that the programmer has not yet known. If the programmer finds one of the delivered methods interesting, a click will take him or her to the full document of the library method in the external documentation system (Javadoc) for Java libraries.

### *4.1.2 Accessing Contextual Information*

If the programmer finds that documentation does not provide enough information to learn to use the method, he or she can click the *Example* link that the SIJ system has added to the standard Javadoc system. This link will take the programmer to example programs that show how the library method is used in other programs (see [36] for details). If the programmer still has questions about using the library method, he or she can click the *Discussion Archive* link, which will take him or her to the archived discussion that shows the questions that other programmers have asked before about the method and the answers provided by experts in the SIJ system.

### 4.1.3 Asking the Experts

If the discussions in the Discussion Archive still do not provide all the information that the programmer needs to learn to use the library method, he or she can click the *Ask Expert* link, which will present a question-posting interface in which the programmer can compose a question. After he or she presses the *Submit* button, the question is sent to a group of expert programmers, who are identified by SIJ. However, the programmer does not know to whom the question is sent. The peer programmers who are identified as "experts" for this particular programmer asking a question on this particular method, and who will receive the question, are determined through two subsequent processes: expert identification and expert selection.

### 4.1.4 Identifying the Experts of the Method

Identification of the peer programmers who are experts on a method is based on the technical profile of each programmer. The technical profile of a programmer in SIJ includes all the library methods in which the programmer has expertise. All the programmers whose technical profiles include the method on which the information-seeking programmer is interested are added to the list of identified experts and become candidates to receive the question. The SIJ technical profile is initiated by parsing the Java programs that the programmer has written and counting the number of usages of each library method [36].

The programmer can also edit his or her own technical profile through the *technical profile management interface* in the SIJ system to indicate a preference of being treated as an expert on a particular task regardless of actual expertise. For example, if a programmer has expertise about certain methods and is willing to share this with other programmers, even though he or she has never used it, the programmer can choose to declare himself or herself as an *Expert* in the technical profile management interface. More important, this interface also allows a programmer to be excluded from receiving questions on methods that he or she does want, even though he or she may know a lot about them. For example, a programmer might deem it wasting time to repeatedly answer questions on methods that do not matching his or her level of skills, instead preferring more intellectually challenging questions. A programmer can declare himself or herself as *No Knowledge* regarding those methods in the technical profile management interface.

Given the huge number of library methods, it is very time-consuming for a programmer to make a declaration on all the methods one by one. To make it easier for programmers to control what questions they prefer to answer, the SIJ system provides the following support. Whenever a question on a method is sent to a selected expert programmer, the programmer can change his or her expertise to *No Knowledge* on that method by clicking on a link embedded in the question that takes the programmer to his or her technical profile management interface.

### 4.1.5 Selecting the Experts

From the list of identified experts, the SIJ system selects those experts who are most likely to answer the question based on the social profiles of each of the experts.

The social profile of each programmer in SIJ has the following attributes: IPP(X, Y), IPO(X, Y), and TSO(X), which represent the *Inter-Personal Preference* of the programmer X toward Y, the *Inter-Personal Obligation* of X toward Y, and the *Total-Social Obligation* of X, respectively.

IPP(X, Y) has three values: 1, 0, and -1, representing, respectively, that programmer X is strongly willing to help programmer Y at all occasions, X is neutral toward helping Y, and X favors not to help Y if possible. The default value of IPP(X, Y) is 0 and can be edited by X at any time through the social profile management interface of SIJ. Similar to the way to manage the technical profile, whenever a question asked by information-seeking programmer Y is emailed to a selected expert programmer X, expert programmer X can click on a link embedded in the email and change his or her IPP(X, Y) to either 1 or -1, declaring that he or she will always answer any questions posted by Y that fall in his or her expertise, or excluding him or her from being selected to answer any of Y's questions. Because the value of IPP(X, Y) is hidden from all other programmers, programmer X can use a socially acceptable way of refusing to help Y by secretly "closing the door" to Y.

IPO(X, Y) is calculated by subtracting the number of times that programmer X has helped programmer Y from the number of times that Y has helped X. A positive value of IPO(X, Y) indicates that X should help Y more to return the direct favors that he or she has received from Y.

TSO(X) is calculated by subtracting the number of times that programmer X has helped others in the group from the number of times that X has received help from others. A positive value of TSO(X) means that X has social obligations to help any other members in the group.

The expert selection process goes through the following five passes:

*Pass 1:* For each member E in the list of the identified experts for the purpose of helping information-seeking programmer A, SIJ first looks at IPP(E, A). If IPP(E, A) is -1, it removes E from the list of identified experts. If IPP(E, A) is 1, it adds E to the list of selected experts and removes E from the list of identified experts.

*Pass 2:* For each member E in the remaining list of the identified experts, if IPO(E, A) is positive, it adds E to the list of selected experts and removes E from the list of identified experts.

*Pass 3:* For each member E in the remaining list of identified experts, if TSO(E) is positive, it adds E to the list of selected experts and removes E from the list of identified experts.

In the beginning of deploying the SIJ system, or when a new member joins the SIJ system, because there is not enough history of social interaction, it is often impossible to select enough experts using the above three passes. To jump-start the social interaction, the SIJ system utilizes the existing social interaction history reflected in each member's email archives. SIJ uses Email(E, A) to represent the number of emails that E has sent to A, determined by mining the email archive of A. If the previous three passes do not result in the selection of a predefined number of experts (the current default setting is 5, which is customizable), it goes further through the following two passes.

*Pass 4:* If the list of selected experts does not reach the predefined number of experts, from the remaining list of the identified experts, SIJ adds experts, according to the order of Email(E, A) to the list of selected experts, until the predefined number is met. The rationale is that the more emails E has sent to A, the more likely E knows A well and offers help to A.

*Pass 5:* If the selected list still does not reach the predefined number of experts, from the remaining list of the identified experts, SIJ adds experts randomly to the list of selected experts until the predefined number is met. This pass is meant to be the

last resort to jump-start the whole information-sharing practice and expand the social relation of those who do not have many interactions with others.

#### *4.1.6 Creating the Socially Aware Communication Channel*

The selected experts will become the members of the ephemeral mailing list and receive the question posed by the information-seeking programmer A. As described in Section 3.3.5, the ephemeral mailing list is a communication channel through which the question posed by the information-seeking programmer is automatically routed to the identified peers without revealing who the recipients are unless they respond to the message.

Because only the recipient knows that he or she received the question, nobody would know if he or she does not offer help. If recipient E does not want to receive more questions on the method, he or she can click the link mentioned in Section 4.1.4 to secretly change his or her expertise on this method to *No Knowledge*. If recipient E has an individual preference to help A, E can also click the link mentioned in Section 4.1.5 to change his or her IPP(E, A).

If recipient E decides to help A, he or she can reply to the question, and the reply is sent to all other members of the ephemeral mailing list, and E's identity is revealed to acknowledge his or her good behavior. At the same time, the social profile of E and A is also updated to reflect the fact that E has offered help to A. IPO(E, A) is reduced by 1 and IPO(A, E) is increased by 1, meaning that E now has less inter-personal obligation toward A, whereas A has more inter-personal obligation toward E. TSO(E) is reduced by 1 and TSO(A) is increased by 1, meaning that E has less total social obligation toward the group, but A has more total social obligation.

When the information-seeking programmer deems there is no more need for further discussions about his or her question because either enough help has been received or the question is not likely to obtain further answers, the programmer should close the ephemeral mailing list. If he or she does not close the ephemeral mailing list after the mail exchange has ceased for a predefined period of time, the ephemeral mailing list will be automatically closed by the system.

The discussion that takes place in the ephemeral mailing list is archived in the discussion archive of the SIJ system and linked to the method. Other members who are not included in the ephemeral mailing list can still have the chance of learning from this information exchange by accessing the discussion archive. The learning benefits of peripheral and passive participation in the traditional broadcasting-to-all mailing list are still retained.

### 4.2 Assessing the SIJ System

This section briefly assesses how the SIJ system follows the principles of providing socio-technical support for Java programmers, as listed in Section 3.1. The SIJ system provides in situ support because the system is embedded in the existing programming environment. Programmers can go directly from their programming practice to obtain help from peer programmers without conscientiously switch from the mental mode of working to the mental mode of learning. Because experts are selected based on the programmer's personal social network, the support is highly individualized: The same question asked by different programmers will result in different groups of experts who receive the question. Furthermore, because the ephemeral mailing list is dynamically created upon the posting of a question, it situates the socio-technical support directly in the constantly changing landscape of the distribution of expertise and the social dynamics of the group. If the same programmer asks the same question at different times, the recipients of the question might be different because of the changes of expertise and social relations of programmers.

Programmers cannot directly ask experts without at least spending some time exploring the existing information accumulated in the SIJ system. The information space of SIJ gradually evolves by archiving the discussion that takes place in the ephemeral mailing list. Such archived discussions are linked to the relevant methods. This design strategy also reduces the number of repeated questions asked and therefore the number of interruptions to the peer programmers.

The number of interruptions is further reduced by excluding irrelevant programmers (either no expertise or no willingness to answer) from receiving the question. The expert identification process ensures that only experts are sent the question, increasing the quality of the answers. The expert selection process considers the willingness of each recipient, increasing the possibility that the programmer can receive a good answer in a timely fashion.

The socially aware communication mechanism (i.e., the ephemeral mailing list) and user-editable technical profile and social profile combine to give experts the full freedom of opting to engage only in information sharing that interests them, with those they prefer, and at their own convenience.

## 5. RELATED WORK

Helping programmers to acquire external information for their programming task has attracted a lot of attention from researchers in software engineering. The STeP_IN framework is an attempt to synthesize the existing techniques and insights into a general framework for creating better supporting tools and environments.

The Hipikat system [8] views the combination of code and documents as an information space from which programmers can glean expertise from existing code and development documents alike with the help of their proposed recommendation technique that mines and extracts hidden relations between code and documents. We further this perspective and conceptualize a software project as an interlinked socio-technical information space by including programmers as information resources, based on their relations with code and documents.

The concept of programmers as information resources has been observed in empirical studies [4] and [20]. The latter cited study has led to the development of the Expertise Recommender system [21], which mines configuration management logs to identify experts and recommend experts explicitly based on organizational relations. The approach of identifying experts from project history was further improved and validated in the Expertise Browser system [23]. Such techniques could be used in instantiated STeP_IN systems that support software maintenance. Again, STeP_IN is a framework that would extend them by taking into account social factors and using existing social relationships.

Some recent studies visualize the complicated interrelation among code, documents, and developers. Ariadne [10], for instance, visualizes social dependency among programmers based on their authorship of interdependent code. Augur [9, 14], simultaneously visualizes the structure of a software system and that of the development process carried out by developers. Hybrid Networks

[22] integrates links from multiple development data sources, such as email archives, CVS source codes, code tree branches, and developers, and uses the Probabilistic Latent Semantic Indexing clustering technique to associate and cluster them. Those research attempts buttress our conceptualization of software projects as a socio-technical information space. Although the main thrust of the visualizations are to reveal the intricate complexity of a software project, they can also be used to aid programmers in determining and locating relevant external information. Furthermore, these techniques used to extract the links for visualization can be applied to the STeP_IN framework for the creation of technical and social profiles.

Help support systems in general have been extensively studied in the Computer Supported Cooperative Work (CSCW) field, of which Answer Garden [1] and its next version Answer Garden 2 [2] are two systems that have developed many features that are folded into the STeP_IN framework. For example, both systems integrate the support of seeking external information from an information repository and peers in a unified way, which is one of the guiding principles of STeP_IN. Answers from peers are all accumulated in discussion archives to reduce the overall cost of answering by avoiding repeated questions. Social relation is also considered in Answer Garden 2 in terms of organizational structure, but not the nuanced perception of individual relationships. The escalation of support, proposed in Answer Garden 2, is in the control of the information-seeker and not the information-provider. This has been the common approach to a number of help support systems developed in the CSCW field, such as Living Design Memory [30] and ContactMap [25]. They disproportionately focus on providing benefits for the information-seeker, without paying much attention to the various burdens incurred upon the information-providers and the negative impacts on the overall productivity of the group to which both information seekers and information providers belong. The STeP_IN framework takes a different direction by focusing more on reducing the burdens to the information provider, and believing sustainable information-sharing acts require the prudent use of the time, attention, and good will of information providers. It does not focus on the success of one information-sharing act in isolation; rather, it treats one act as an episode that is situated in the context determined by the information being sought and the social dynamics among the group members, and that also shapes the context for future acts.

# 6. DISCUSSIONS

The effectiveness of the STeP_IN framework relies heavily on the accuracy and completeness of the technical and social profiles of each programmer. Accurate and complete profiles require much effort from programmers. We have designed mechanisms of amortizing the efforts into smaller steps (i.e., direct links from question emails for changing profiles). Still, the collective cost cannot be neglected. In most situations, the framework has to work with proximate profiles automatically generated from historical data, and this would cause inaccuracy in identifying and selecting experts. Consequentially, some able and willing experts might be left out of the selection, and the information-seeking programmer might not be able to obtain help that he or she would receive if the profiles were accurate and complete. One possible solution is to apply the idea of escalation of support [2]. When no answers are provided from the selected group for a predefined period of time, the system automatically expands the recipients of the question to all experts and finally to the whole group.

One may inquire why we do not ask all the experts or all the members through mailing lists in the first place, or provide the list of experts and simply let the programmer choose a member to ask through direct emails. Mailing lists and direct emails certainly have their advantages. In mailing lists or bulletin board systems, repliers have complete control over when to answer the problem and the freedom of no action because questions are not directly addressed to them. The information seeker, however, has no way to control the quality of the answers and push for an answer. Furthermore, all members, no matter whether they have an interest in the question or not, are interrupted to different degrees. Although the cost for each member caused by one question is rather small, when we multiply that cost by the number of members and the number of questions, the total cost becomes quite large, and its impact on group productivity is big. In contrast, in direct mails, the receiver loses control of collaboration and bears the social burden and interruption cost of reaction or no action [31].

The STeP_IN framework attempts to find an alternative way that is geared toward the specific needs of the type of ad hoc and in situ information gathering from peer programmers. It is certainly not meant to replace direct emails and mailing lists for other communication needs.

# 7. CONCLUDING REMARKS

The STeP_IN framework sees peer programmers as important resources for information because much of the information of a software project remains in the heads of programmers. It thus conceptualizes a software project as a socio-technical information space consisting of code, documents, and programmers that are interrelated. The framework provides incremental support of information acquisition from immediately relevant information, to contextual information, and to peers as needed.

Programming is social practice. Most existing programming environments, however, are designed to support only the technical aspects of programming for a single programmer. Programmers mostly have to rely on generic and stand-alone communication tools to meet their needs for information sharing that arise from programming tasks. The STeP_IN framework is an initial attempt to systematically treat both technical issues and social issues in programming support. We envision the framework to serve as a starting point for further investigations in this direction, and as a generic framework that can be enhanced and improved by other researchers and ourselves through instantiating it into different systems.

# 8. REFERENCES

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. We would like to thank Kouichi Kishida, Yoshiyuki Nishinaka, and , Mitsuhiro Asada for their valuable contributions for the development of the work presented in the paper.

# 9. 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. 31-39.

[2] Ackerman, M.S. and D.W. McDonald, Answer Garden 2: Merging Organizational Memory with Collaborative Help, in *Proceedings of CSCW'96*. 1996. 97-105.

[3] Antoniol, G., et al., Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 2002. **28**(10): 970-983.

[4] Berlin, L.M., Beyond Program Understanding: A Look at Programming Expertise in Industry, in *Empirical Studies of Programmers: Fifth Workshop*, 1993, Ablex Publishing Corporation: Palo Alto, CA. 6-25.

[5] Card, S., G. Robertson, and J. Mackinlay, The Information Visualizer: An Information Workspace, in *Proceedings of CHI'91*. 1991. 181-188.

[6] Cosley, D., et al., Using Intelligent Task Routing and Contribution Review to help Communities Build Artifacts of Lasting Value, in *Proceedings of CHI'06*. 2006. 1037-1046.

[7] 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, Eds. 2004, The MIT Press: Cambridge, MA. 137-161.

[8] Cubranic, D. and G.C. Murphy, Hipikat: Recommending Pertinent Software Development Artifacts, in *Proceedings of ICSE03*. 2003. 408-418.

[9] de Souza, C., J. Froehlich, and P. Dourish. Seeking the Source: Software Source Code as a Social and Technical Artifact, in *Proceedings of GROUP05*. 2005. 197-206

[10] de Souza, C., et al., How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development, in *Proceedings of FSE04*. 2004. 221-220.

[11] DeMarco, T. and T. Lister, *Peopleware: Productive Projects and Teams*. 2nd ed. 1999, New York: Dorset Housing Publishing.

[12] Fischer, G. and K. Nakakoji, Making Design Objects Relevant to the Task at Hand, in *Proceedings of Ninth National Conference on Artificial Intelligence*. 1991. 67-73.

[13] Fischer, G., E. Scharff, and Y. Ye, Fostering Social Creativity by Increasing Social Capital, in *Social Capital*, M. Huysman and V. Wulf, Eds. 2004, MIT Press: Cambridge, MA. 355-399.

[14] Froehlich, J. and P. Dourish, Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams, in *Proceedings of ICSE04*. 2004. 387-396.

[15] Fukuyama, F. Social Capital and Civil Society, in *IMF Conference on Second Generation Reforms*. 1999. Washington, DC.

[16] Herbsleb, J. and A. Mockus, An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, 2003. **29**(3): 1-14.

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

[18] Kraut, R.E., et al., Social Impact of the Internet: What Does it Mean? *CACM*, 1998. **41**(12): 21-22.

[19] LaToza, T.D., G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits, in *Proceedings of ICSE06*. 2006. 492-501

[20] McDonald, D.W. and M.S. Ackerman, Just Talk to Me: A Field Study of Expertise Location, in *Proceedings of CSCW'98*. 1998. 315-324.

[21] McDonald, D.W. and M.S. Ackerman, Expertise Recommender: A Flexible Recommendation System Architecture, in *Proceedings of CSCW'00*. 2000. 101-120.

[22] Medynskiy, Y., N. Ducheneaut, and A. Farahat, Using Hybrid Networks for the Analysis of Online Software Development Communities, in *Proceedings of CHI06*. 2006. 513-516.

[23] Mockus, A. and J. Herbsleb, Expertise Browser: A Quantitative Approach to Identifying Expertise, in *Proceedings of ICSE02*. 2002. 503-512.

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

[25] Nardi, B.A., S. Whittaker, and H. Schwarz, It's Not What You Know, It's Who You Know: Work in the Information Age. *First-Monday: Peer-reviewed Journal on the Internet*, 2000. **5**(5).

[26] Pentland, A., Socially Aware Computation and Communication. *Computer*, 2005. **38**(3): 33-40.

[27] Perlow, L.A., The Time Famine: Toward a Sociology of Work Time. *Administrative Science Quarterly*, 1999. **44**(1): 57-81.

[28] Robillard, P.N., The Role of Knowledge in Software Development. *CACM*, 1999. **42**(1): 87-92.

[29] Szoestek, A.M. and P. Markopoulos, Factors Defining Face-to-Face Interruptions in the Office Environment, in *Proceedings of CHI06*. 2006. 1379-1384.

[30] Terveen, L.G., P.G. Selfridge, and M.D. Long, Living Design Memory: Framework, Implementation, Lessons Learned. *Human-Computer Interaction*, 1995. **10**(1): p. 1-37.

[31] Tyler, J.R. and J.C. Tang, When Can I Expect an Email Response? A Study of Rhythms in Email Usage, in *Proceedings of the Eighth European Conference on Computer Supported Cooperative Work*. 2003. 239-258.

[32] Westrup, C., On Retrieving Skilled Practices: The Contribution of Ethnography to Software Development, in *Social Thinking: Software Practice*, Y. Dittrich, C. Floyd, and R. Klischewski, Eds. 2002, MIT Press: Cambridge, MA. 95-110.

[33] Ye, Y. Information-Enriched Workspaces, in *Proceedings of IFIP 2001 International Conference on Human-Computer Interaction*. 2001. Tokyo, Japan: IOS Press. 206-213

[34] Ye, Y. and G. Fischer. Information Delivery in Support of Learning Reusable Software Components on Demand, in *Proceedings of 2002 International Conference on Intelligent User Interfaces*. 2002. 159-166.

[35] Ye, Y. and G. Fischer, Supporting Reuse by Delivering Task-Relevant and Personalized Information, in *Proceedings of ICSE02*. 2002. 513-523.

[36] Ye, Y., et al., Searching the Library and Asking the Peers: Learning to Use Java APIs on Demand, in *Proceedings of 2007 International Conference on Principles and Practices of Programming in Java*. 2007. (forthcoming)

[37] Zimmermann, T., et al., Mining Version Histories to Guide Software Changes, in *Proceedings of ICSE04*. 2004. 563-572.