Expanding the knowing capability of software developers through knowledge collaboration

Yunwen Ye*

SRA Key Technology Laboratory, Inc.
3–12 Yotsuya, Shinjuku, Tokyo 160–0004, Japan
and
Center for LifeLong Learning and Design
Department of Computer Science
University of Colorado
Boulder, CO 80309–0430, USA
E-mail: yunwen@colorado.edu
*Corresponding author

Yasuhiro Yamamoto

Research Center for Advanced Science and Technology
University of Tokyo
4–6–1 Komaba, Meguro, Tokyo, 153–8904, Japan
E-mail: yxy@kid.rcast.u-tokyo.ac.jp

Kumiyo Nakakoji

SRA Key Technology Laboratory, Inc.
3–12 Yotsuya, Shinjuku, Tokyo 160–0004, Japan
and
Research Center for Advanced Science and Technology
University of Tokyo
4–6–1 Komaba, Meguro, Tokyo, 153–8904, Japan
E-mail: kumiyo@kid.rcast.u-tokyo.ac.jp

Abstract: Because software development is a knowledge-intensive process, the support of software developers presents two equally important challenges: the establishment of a rigorous and quantifiable foundation for software systems, and a better understanding of knowledge creation processes that take place in software development. Software engineering research has traditionally focused mostly on the former challenge. Focusing on the latter challenge, this paper conceptualises software development as a knowledge-intensive and collaborative activity. This new conceptualisation leads to the argument that knowledge collaboration plays a central role in software development. The paper discusses different forms of knowledge collaboration in software development, roles that computers can play to support knowledge collaboration, and the associated technical and social challenges. It argues for the need to provide *in situ* and individualised support for knowledge collaboration through the approach of *layered support on demand*, which is illustrated with systems that we have developed.

Keywords: knowledge collaboration; software development support; sociotechnical environment; layered support on demand.

Reference to this paper should be made as follows: Ye, Y., Yamamoto, Y. and Nakakoji, K. (2008) 'Expanding the knowing capability of software developers through knowledge collaboration', *Int. J. Technology, Policy and Management*, Vol. 8, No. 1, pp.41–58.

Biographical notes: Yunwen Ye is a Senior Researcher at SRA Key Technology Laboratory, Inc., Japan, and a Research Associate at the University of Colorado, Boulder. His research interests include the cognitive and social aspects of software development, open-source software systems, computer support for knowledge-intensive work and human-computer interaction.

Yasuhiro Yamamoto is an Associate Professor at the Research Center for Advanced Science and Technology (RCAST), University of Tokyo, where he co-directs the Knowledge Interaction Design (KID) laboratory. He has worked as a Post-doctoral Researcher at the Japan Society for the Promotion of Science, Japan Science and Technology Corporation, and RCAST. His research interests include human-computer interaction, interaction design and design process design.

Kumiyo Nakakoji is a Full Professor at RCAST, University of Tokyo, and a Senior Research Fellow at SRA Key Technology Laboratory, Inc., Japan. She has served as Chair, Editor, and committee member for a number of research journals and conferences in such fields as human-computer interaction, software engineering, and collective creativity support. Her current research interest is knowledge interaction design, which is a framework for the design and development of computational tools for creative knowledge work.

1 Introduction

Software development is knowledge intensive because it involves the application of knowledge that comes from multiple domains. Software developers, therefore, are knowledge workers, engaging in what Lazzarato (1996) called *immaterial labour*, which is defined as the labour that produces the informational and cultural content of a commodity. Immaterial labour blurs the traditionally sharp division between conception and execution, creativity and labour, intellectual labour and material labour.

Software development is also a social practice. The process of software development is enabled by intense interactions among software developers. Most systems involve hundreds of developers who have to communicate and collaborate with each other in both formal and informal ways.

The research on understanding and supporting software development has been carried out mainly under the umbrella of software engineering. The primary concerns of software engineering have been the establishment of quantifiable scientific foundations towards the ultimate goal of reaching the stage where software systems can be built in a methodical way through a specific and well-defined sequence of steps. Thus, the productivity and quality of software development can be predicted and dictated mainly by the soundness of the approach rather than by the craft and creativity of individual developers. Such a focus seeks universal 'best practices' in software development regardless of the nature of the software, the individuality of developers, and the sociotechnical situations of the group of developers.

Although software development does share many characteristics with other engineering disciplines, it presents new research challenges because the whole lifecycle of software production and use is a kind of immaterial labour that critically hinges on the knowledge and creativity of individual software developers and their intense social interactions. In addition to focusing on the engineering aspects of software development, we also need to think about the ramifications of software development as a goal-oriented cognitive activity and a creative and social practice. We thereby need to focus on the cognitive, creative, and social aspects of software development and software developers. This alternative perspective aligns software development more with writing, design, and architecture in which individual differences and creativity are more valued and encouraged.

The writings of Weinberg (1971), Brooks (1987) and others (*e.g.*, Fischer, 2002) have touched upon the importance of many issues associated with the human-centric viewpoint of software development, but unfortunately the attention given by software researchers to the engineering aspects and human aspects of software development are overwhelmingly disproportionate. From this human activity perspective, this paper attempts to conceptualise software development as a knowledge-intensive and distributed cognitive activity (Hollan *et al.*, 2001), and ponders its implications for developing tools that support software development.

The knowledge required for the creation of complex software systems is immense. Given the limitations of human memory, learning capability, and time dedicated to learning, few developers, if any, have all the knowledge needed in their own heads. The knowledge is distributed between the developer and the external world, and the development of a software system requires that a software developer not only fully utilise the knowledge in the head but also exert his or her *knowing capability* – the ability to access and learn from various knowledge resources in the world – in the context of development.

Knowledge repositories and knowledgeable peers have been regarded as the two primary important external knowledge resources. To expand his or her knowledge capability, a software developer therefore needs to engage in an activity that we call *knowledge collaboration* – utilising knowledge repository systems as well as enlisting the help of peer developers to acquire the needed knowledge that the developer does not have yet. This paper identifies research questions that arise from shifting our focus to the knowing capability of software developers and their demonstrated effective behaviours in practice; develops the framework of knowledge collaboration that can expand knowing capability; and introduces the approach of *layered support on demand*, which supports *in situ* knowledge collaboration by dynamically creating a network of knowing according to who is looking for what, and under what kind of sociotechnical environment.

The paper is structured as follows. Section 2 introduces theoretical foundations that undergird the conceptualisation of software development as a knowledge-intensive and collaborative activity, and elucidates the importance of knowing capability in the practice of software development. Section 3 discusses various aspects of knowledge collaboration in software development and the technical and social challenges therein. Section 4 describes the approach of layered support on demand that overcomes such challenges with a brief description of systems in which the approach has been instantiated. Section 5 presents conclusions.

2 Theoretical foundations

2.1 From engineering to design

Software development is a process of creating representations that can be executed by computers. In addition to the final formal representations (*i.e.*, programs or code), developers also make, during the process of development, many other kinds of representations (often called documents), such as requirements, specifications, design diagrams, and test plans for the purpose of communication and collaboration. Representation making is therefore central to software development, and the centrality of representation-making makes software development an instance of design rather than an instance of manufacturing that ends up with physical artefacts.

A design process consists of a series of goal-oriented cognitive activities in which the goal itself is often not well defined in the beginning and needs to be better defined through the design process itself. Empirical studies of designers in other fields as well as in software development have shown that this design process, contrary to the beliefs of many software methodologists, does not follow any preset procedures (Detienne, 1995). It is dynamically determined by the knowledge of the designer and the feedback from the sociotechnical environment in which the designer works. In other words, the design process is a continuous conversation between human minds and the interim design representations through repeated cycles of action and reflection-in-action (Schön, 1983).

Software developers are therefore reflective practitioners. They act to make representations. Owing to the complexity of software systems, the conflicts of design requirements to be reconciled, and the possibility of various ways of addressing the same problem, those representations are tentative and tend to produce consequences other than those intended by the developers. Developers need to reappreciate their newly created design situations by reflecting upon the feedback presented by the representations, and then act further in an opportunistic way by constantly shifting attention to critical parts of the problem. The skills demonstrated by designers lie in how they are able to take situated actions in response to the situational back-talk through reflection-in-action.

Viewing software development as a series of reflective design activities implies a new set of requirements for software development environments: How can we increase the situational back-talk of developer actions? How can we enrich the development environment with more task-relevant information that stimulates reflection?

2.2 From knowledge to knowing capability

Both representation making and reflection during the software development process require developers to activate and apply knowledge relevant to the task at hand. The notion of ‘knowledge’ has expanded from its traditional focus on knowledge as ‘stored artefacts’ in a person’s head to the focus of a person’s ability to find one’s way around the world through effective interaction and collaboration with tools and people in his or her environment. To achieve efficiency in such knowledge-intensive work as software development, knowledge workers need not only to utilise the knowledge that they have acquired in their heads, but also to exert their knowing capability in practice. This expansive perspective of knowledge leads away from the attention on what a knowledge worker knows in abstract, which has been the major focus of current discourse on software engineering education and knowledge management, and towards a focus on the situated actions taken by developers when they engage in work, and the intelligent behaviours demonstrated by developers in utilising whatever internal and external resources are available in practice (Orlikowski, 2002).

The concept of knowing capability in practice is grounded in the theoretical thinking of philosopher Polanyi (1966), who noted that most of human knowledge is tacit, and sociologist Giddens (1984), who argued that immense knowledgeability is involved in the effective actions of human beings. This type of theoretical thinking is further supported by the anthropological studies of Hutchins (1994) and Suchman (1987).

Shifting the focus from individual knowledge in the abstract to knowing capability in the context of human interaction with tools and peers has led to the development of the theory of distributed cognition, which views an individual and the surrounding context as a whole cognitive system (Hollan *et al.*, 2001) to be analysed and supported. Hollan *et al.* point out that cognitive processes are distributed along at least three dimensions:

1 Cognitive processes may be distributed across the members of a social group.

2 Cognitive processes may involve coordination between internal and external (material or environmental) structures.

3 Cognitive processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events.

Focusing on the knowing capability of software developers in practice, we can pose the following research challenges: What are the suitable configurations of developers, representations, and tools that can enhance the knowing capability of software developers? How can we create appropriate sociotechnical platforms that support software developers in acquiring external knowledge in the world to complement their insufficient knowledge 'in the head'?

2.3 From division of labour to distribution of knowledge

As software systems become more complex and larger, the amount and kinds of knowledge required increase at a rapid speed and change constantly. For example, application domains are subject to rapid changes, a vast amount of third-party libraries are continually updated, new features and functionalities continue to be introduced in programming tools and environments, and new languages are invented. The required body of knowledge for the development of modern software systems is becoming so vast that no single developer can have all the knowledge (Robillard *et al.*, 2004).

The knowledge required in software development includes not only the process knowledge and domain knowledge that are distilled into the software system, it also includes knowledge about the software system itself that developers are currently creating. One may argue that because the software developer participates in the creation of the system, he or she should know the system inside out. However, because large-scale software systems are created collaboratively by many people, not all developers, if any, would have complete knowledge about the whole system. At the same time, with the increasingly widely accepted view of software systems as evolving entities, the percentage of incremental, continuous development tasks in software development has risen quickly (Rajlich and Gosavi, 2004). Such software systems need to be continuously developed with iterative processes. In addition, with the high turnover rate in the software industry, many developers are working to make incremental changes to systems that have been partially developed, or even to systems that are already operating daily.

Software developers in a project team, therefore, do not have a uniform knowledge structure. Each of them has a unique set of skills and expertise, and at the same time does not know all of what other developers know, leading to the phenomenon of 'asymmetry of knowledge' (Fischer, 1999). This asymmetry of knowledge is not a barrier; rather, it is an opportunity that needs to be explored fully to enhance the *collective capability* (Orlikowski, 2002) of a project team. The key is how to integrate this diversity of expertise and synthesise it into the collective knowing of a software project team through collaboration in which ideas and inspirations cross fertilise and feed on each other.

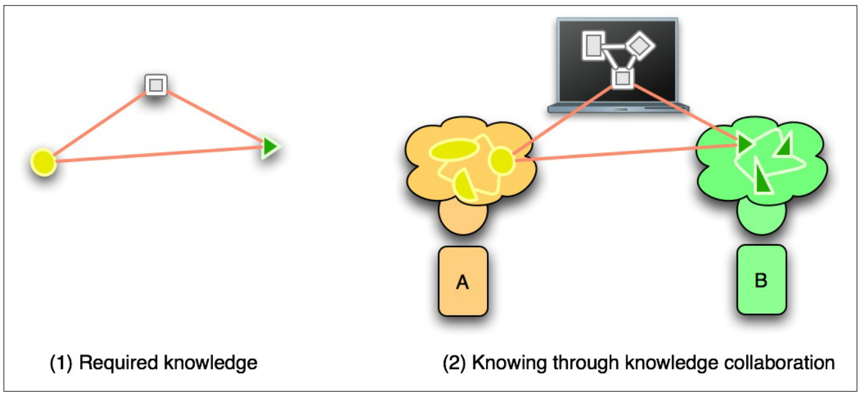
Collaboration has been a major research topic in software engineering. However, most of the current research has focused on the brawny aspects of collaboration, namely, the power brought by many hands. The major concerns have been on the cooperation, communication, and coordination problems brought about by the consequences of division of labour (Herbsleb and Mockus, 2003). The other aspect of collaboration in software development in need of more research attention is the brain power of the multiple heads of developers, namely, the collective knowing capability brought forth by the unique knowledge of developers who mutually complement each other.

3 Knowledge collaboration in software development

Knowledge plays a central role as software developers engage in reflective practices in software development, but software developers do not always have all the knowledge that is required for their tasks because knowledge is distributed among developers. Many empirical studies of software development have shown that software developers routinely engage in seeking external expertise (McDonald and Ackerman, 1998). Expert developers demonstrate knowledgeable behaviours not only because they have more knowledge than novice developers but also because they are able to obtain help from other developers in a timely fashion (Berlin, 1993) and to find and utilise external knowledge resources more effectively.

When a developer does not have sufficient knowledge for his or her development task, that developer must engage in *knowledge collaboration* by enlisting the help of external knowledge resources that either are embedded in computational tools or exist only in knowledgeable peers. Knowledge collaboration, a joint intellectual endeavour in which a software developer engages with external cognitive tools and knowledgeable peers, has thus become an essential activity in most of today's software development, and it defines the knowing capability and competence of software developers. Figure 1 illustrates the core idea of knowledge collaboration. Suppose a development task requires a combination of three knowledge elements, as shown in Figure 1–1; and Developer A, who is in charge of the task, has only one element in his or her head, and the other two elements are held in a tool and by Developer B. However, if Developer A is able to construct a *network of knowing*, as shown in Figure 1–2, enlisting the help of the tool and Developer B, Developer A would be capable of accomplishing the task just as if he or she possessed all the knowledge elements in his or her own head.

Figure 1 Knowledge collaboration



3.1 Forms of knowledge collaboration

Following Nahapiet and Ghoshal (1998), who argue that all new resources, including knowledge, are created through the two generic processes of transfer and combination, we view knowledge transfer and collaborative knowledge construction as two basic forms of knowledge collaboration.

Knowledge transfer takes place when knowledge held by one person is transferred to others. Knowledge transfer does not necessitate direct interactions if knowledge can be codified and stored in knowledge repositories, from which the knowledge seeker is able to retrieve and apply it to his or her own work. However, the other major form of knowledge, tacit knowledge, cannot be readily transferred through knowledge repositories. Because tacit knowledge is essentially embedded in everyday practices and social interactions, it can be transferred only through social interactions. In such cases, direct communication and interaction between knowledge seekers and knowledge providers have to take place for the tacit knowledge to become transferable.

Collaborative knowledge construction occurs when knowledge workers cannot cover the full spectrum of knowledge required to solve a problem. New knowledge has to be created by combining knowledge held by each worker. Collaborative knowledge construction is not a simple sum of each participant's knowledge, but a result of mutually stimulating exchange embedded in social interactions and practices. During this process, implicit assumptions are externalised; different viewpoints are presented, negotiated, and integrated; and formerly unconnected knowledge elements get connected and combined to create new knowledge.

3.2 Dimensions of knowledge collaboration

Knowledge collaboration can take place along two axes: the technological axis and the social axis. Along the technological axis, knowledge workers utilise external cognitive tools to complement the insufficient knowledge in their heads. Cognitive tools include books, manuals, and computerised information repositories such as online help systems and reuse repository systems. Knowledge collaboration supported by external cognitive tools facilitates mainly indirect knowledge transfer, and the combination of knowledge is conducted in the head of the knowledge seeker. However, as researchers in knowledge management have repeatedly discovered, the mere existence of a knowledge repository does not naturally lead to the transfer of knowledge (Fischer and Ostwald, 2001). To utilise external knowledge from repositories, software developers are faced with a number of cognitive challenges (Ye and Fischer, 2002):

- They may not even be aware of the existence of useful external knowledge.
- They may not be able to find the relevant external knowledge.
- They may not be able to understand and apply the new knowledge.
- Different software developers have different knowledge needs and require individualised support.

When perusing the external knowledge itself is not enough for learning, due to the inability to capture contextual and tacit knowledge in knowledge repositories, software developers often turn to knowledgeable peers for help (Berlin, 1993). Knowledge collaboration, hence, also takes place along the social axis by enlisting the help of peers. The basic challenges in supporting knowledge collaboration along this social axis are:

- Software developers may not know to whom they can turn for help on a particular problem (Becks *et al.*, 2004).
- Experts who are able to help may not be willing to do so owing to the interruption to their own work, among various other reasons (Cross and Borgatti, 2004).

3.3 Roles of computers in supporting knowledge collaboration

Computers can play two roles to support knowledge collaboration in software development:

1 as a cognitive tool to support software developers

2 as a mediating tool to provide a platform that facilitates knowledge collaboration among software developers.

3.3.1 Computer-augmented knowing capability

Computer-based tools such as compilers, editors, and reusable libraries have always been an important part of the repertoire that constitutes the technical competence of software developers. However, such tools have not become equal partners in the underlying knowledge-intensive process of software development. Developers use such tools by delegating a part of development tasks such as compiling and linking, or by utilising them to carry out a planned action such as editing. Equal partners in the knowledge collaboration process should be able to directly contribute their own knowledge to the development process by augmenting the back-talk of user actions in order to complement the insufficient knowledge of software developers and to exert direct influences on subsequent reflections and actions of developers. With the intelligence augmentation brought by computer systems, software developers are able to demonstrate a knowledge capability that is larger than their knowledge in the head would have enabled.

Automatic spelling correction in a word processor is one simple example of enhanced knowing capability. If a student fails to spell an English word correctly, an English teacher would view the student as lacking knowledge of that word. However, if the student is writing an essay with a word processor and ventures a guessed spelling that is close enough, the automatic spelling correction would be able to help the student spell correctly. Readers of the essay would then think the student has knowledge of the word.

An example of research that views computers as equal partners in the knowledge collaboration process of software development is the LispCritic system (Fischer, 1987), which is an intelligent agent that helps programmers improve their programming skills. It suggests better solutions after it has recognised a programmer's less ideal code segment. Programmers can improve their skills by working together with the LispCritic system that provides feedback.

3.3.2 Computer-mediated knowledge collaboration

In computer-mediated knowledge collaboration, the computer plays the role of connecting software developers in need of new knowledge with developers who have the knowledge. Computational environments provide sociotechnical platforms for software developers to exchange and combine knowledge from different people. This line of research has been conducted mainly in Computer-Supported Cooperative Work (CSCW).

Because software development has traditionally been viewed as an individual cognitive process in software engineering, there has not been much work in applying CSCW research results to support software development. The Answer Garden system (Ackerman and Malone, 1990) is an early effort to support Unix users in obtaining help from other knowledgeable peers by routing questions asked by a user to pre-registered Unix experts. Expertise Browser (Mockus and Herbsleb, 2002) mines configuration management logs to find who has expertise on a portion of a system.

The above systems have focused on finding the experts on a particular problem or a domain. An important aspect that has been missing is why the experts should participate in the knowledge collaboration process. For a knowledge collaboration process to succeed, finding who the experts are is only the first step. As knowledge resources, 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" (Illich, 1971). To create a computational platform to support knowledge collaboration, the motivation to participate has to be considered from the beginning.

3.4 Technical support and social support

The above two roles that computers play in supporting knowledge collaboration (as a cognitive tool and a mediating tool) are similar to the two threads of research in knowledge management: the repository-based approach and the community-based knowledge-sharing approach, respectively.

Deeply influenced by research in artificial intelligence, the knowledge repository approach believes that expertise or knowledge can be externalised and formalised for sharing. The research focus, therefore, is on extracting knowledge by interviewing experts, formalising extracted knowledge, storing formalised knowledge in knowledge repositories, and developing retrieval mechanisms to locate and retrieve knowledge from repositories (Fischer and Ostwald, 2001).

Based on a realisation of the difficulties in the knowledge repository approach that cannot capture the tacit knowledge, a community-based approach was proposed (Lave and Wenger, 1991). This approach focuses on understanding and supporting knowledge transfer and collaboration through human communications because it believes that learning takes place naturally when people from a community of practice engage in practices together. In contrast to the knowledge repository approach, in which knowledge is managed, this approach advocates promoting informal communication in communities through the management of social networks and the creation of expert lists, so that users know whom to ask when they have a problem.

Both approaches have their merits and shortcomings. The repository-based approach is fully controlled by the user, who can enlist the collaboration of knowledge repository systems whenever needed. The community-based approach is subject to the time and willingness of other partners. The repository-based approach is more efficient, and the community-based approach incurs considerable cost because it requires users to invest time and effort in advance to establish membership in the community and it also requires the time, attention, and goodwill of knowledgeable peers. The major problem with the repository-based approach, as we have pointed out, is that it cannot capture tacit and contextual knowledge, and therefore its support for knowledge collaboration is limited.

An ideal situation is that whenever the knowledge collaboration can be handled by the repository-based approach, we should strive for that. The community-based approach should be better used as a social backup only when it is needed because social support is more costly. By integrating the two approaches, continuous support can be provided for software developers to engage in knowledge collaboration according to their unique needs.

4 Layered support on demand

The need for knowledge collaboration in software development arises during the software developer's practice and varies according to who is currently doing what. The support for knowledge collaboration therefore has to be contextualised to the particular situation in which the needs for collaboration arise and is individualised to the particular developer who is seeking external knowledge. Different developers have differing needs for external knowledge, even though faced with the same development task, because the background knowledge of each varies. For the same developer, the need for external knowledge may also be different, depending on the current development task.

To provide contextualised and individualised support for knowledge collaboration, we have developed the approach of *layered support on demand*, which divides support for knowledge collaboration into different layers and provides each support layer *in situ* based on the individualised demand of each developer. The approach embeds sociotechnical support for knowledge collaboration in development environments and provides a natural link from the task at hand to a variety of external information and knowledge resources, which are presented at different levels of abstraction in response to the varying needs of each developer. Depending on the working context and background knowledge of each developer, a unique network of information and knowledge is dynamically constructed and presented to provide a seamless transition from developers' interactions with tools to interactions with external knowledge and other knowledgeable peers.

4.1 Layers of sociotechnical support for knowledge collaboration

Successful knowledge collaboration with external knowledge resources requires that software developers be able to locate, comprehend, and use external knowledge in a timely fashion when the need arises during their practice. This process involves at least four different phases: awareness, discernment, evaluation, and application (Carey and Rusli, 1995). In the *awareness* phase, software developers have to be made aware of the existence of external knowledge that can be applied in their current tasks. In the *discernment* phase, software developers quickly determine whether a piece of knowledge is relevant to their current task. The *evaluation* phase involves a detailed study of the knowledge for a relatively thorough understanding. In the *application* phase, software developers need to combine the external knowledge with their existing appreciation of the development situation and apply it in the current task.

Each phase requires a different level of detail. For awareness, software developers need a prompt about the existence of relevant external knowledge. For discernment, an overview of the knowledge is enough. This phase also requires the simultaneous presentation of several candidates for software developers to compare to find the knowledge that is most relevant. For evaluation, software developers need to focus on the details of a chosen piece of knowledge. In the application phase, examples that use the knowledge are very effective in helping users understand, adapt, and integrate the knowledge by providing context and enabling software developers to draw analogies between their task and examples.

The detail required for utilising external knowledge depends on the background knowledge of the developer. Because each software developer has a unique knowledge structure, his or her needs for knowledge collaboration vary. A continuum of demands for different levels of detail exists. On one extreme, if the developer already knows the presented knowledge, even vaguely, an overview of the knowledge may be enough. On the other extreme, if the developer has never encountered the piece of knowledge before, he or she may need to go through all the phases. Owing to the tacit nature of knowledge, a knowledge repository cannot capture all the knowledge, and finding an expert on the topic becomes necessary for the user to understand and apply the knowledge.

Knowledge collaboration with peer experts presents further challenges for individualised support. First, as the notion of asymmetry of knowledge conveys, knowledge is not evenly distributed among software developers. Therefore, the term 'expert' is not an absolute attribute, but it is only a relative attribute of a developer, depending on the knowledge. Experts can be identified only after the piece of knowledge of interest is known. Second, interpersonal social relationships play critical roles in motivating and shaping the ways that people collaborate with each other (Cross and Borgatti, 2004). Upon finding experts, we have to take into consideration the unique social network of each developer.

4.2 Providing individualised support on demand

Using one typical software development activity – software reuse – as an example to illustrate the approach of providing layered support on demand for each software developer's contextualised and individualised needs for knowledge collaboration, this section briefly describes the integration of two systems that we have developed: CodeBroker and STeP_IN. The description of the systems in this paper is not about the systems themselves; rather, it is meant to illustrate how the approach of the continuous layered support on demand can be instantiated. For further details about the goals, functionality, design, implementation, and evaluation of CodeBroker and STeP_IN, please see Ye and Fischer (2002) and Ye *et al.* (2007), respectively.

The CodeBroker system addresses the cognitive challenges in knowledge collaboration along the technological axis: (1) being aware of the existence of external knowledge (Java API library in this case) as well as (2) being able to locate the needed knowledge from a large repository. The STeP_IN system addresses the challenges in the social axis of knowledge collaboration by helping software developers find peer developers who (1) have expertise on the piece of knowledge of current interest and (2) are likely to be willing to offer timely help to the particular developer in need of support. The STeP_IN system is based on a new conceptual framework, DynC (Ye *et al.*, 2004), that we have proposed to support situated knowledge collaboration.

The following subsections describe the six layers of support for continuous knowledge collaboration, with the first two layers supported by CodeBroker and the rest supported by STeP_IN. The problem to be addressed is to help software developers learn to reuse Java API library components on demand that they do not yet know during their development practice. The huge size of the Java API library (*e.g.*, Java SDK 1.5.0 has 3279 classes) contributes greatly to the productivity gain of software developers but also presents significant learning challenges to Java developers. It is hardly possible – but unnecessary as well – for Java developers to learn the whole library before they start development in Java. They have to learn incrementally during their development practice.

4.2.1 Autonomous individualised delivery to raise awareness

The first challenge for software developers to learn unknown components on demand is that they have to be aware of the existence of some library components that can be reused in their current task. With CodeBroker, which is completely integrated with the development environment Emacs, software developers do not need to explicitly start searching for task-relevant library components. As software developers enter document comments and method signatures in the editing space (Figure 2–0), the system extracts the comments and signatures and uses them in a query. A list of task-relevant components (methods) that match the query is autonomously retrieved from the Java API library and is displayed into the delivery buffer (Figure 2–1). This autonomous delivery makes it possible for software developers to use information that they are not even aware exists in the information repository, a well-recognised cognitive barrier to the utilisation of knowledge repositories (Ye and Fischer, 2002).

Different users have different levels of knowledge and need different sets of external knowledge, even for the same task. CodeBroker therefore uses user models to personalise the retrieval results before they are delivered. The retrieved task-relevant components are compared against the developer's user model, which contains the components he or she already knows. The known components are removed because the developer would already be able to reuse those known components if they are reusable in the current situation.

Each delivered component (Figure 2–1) is accompanied by a rank of its relevance to the current task, the relevance value, its name, and the synopsis of its functionality. This is a context-aware list of reusable components for the developer to browse, and it serves as the first layer of support presented on demand. This layer of support increases the situational back-talk of a developer's work environment by making the developer aware of unknown components that are relevant to his or her current development task, making it possible to directly utilise knowledge that is still external.

4.2.2 Mouse movement-triggered information

Another important piece of information for a Java API method is its signature. We decided not show the signature in the delivery buffer because signatures are usually long and take too much of the scarce screen resource. Furthermore, signatures become relevant only when the developer finds the library component has the potential to be reused. In this sense, the signature of a component is secondary information needed to determine the applicability of the component. Therefore, only when the developer moves the mouse pointer over the component names in the delivery buffer (Figure 2–1), the component's full signature is shown in the mini-buffer (Figure 2–2). This is the second layer of information presented on demand to assist the developer in further determining the relevance of the component to the task at hand.

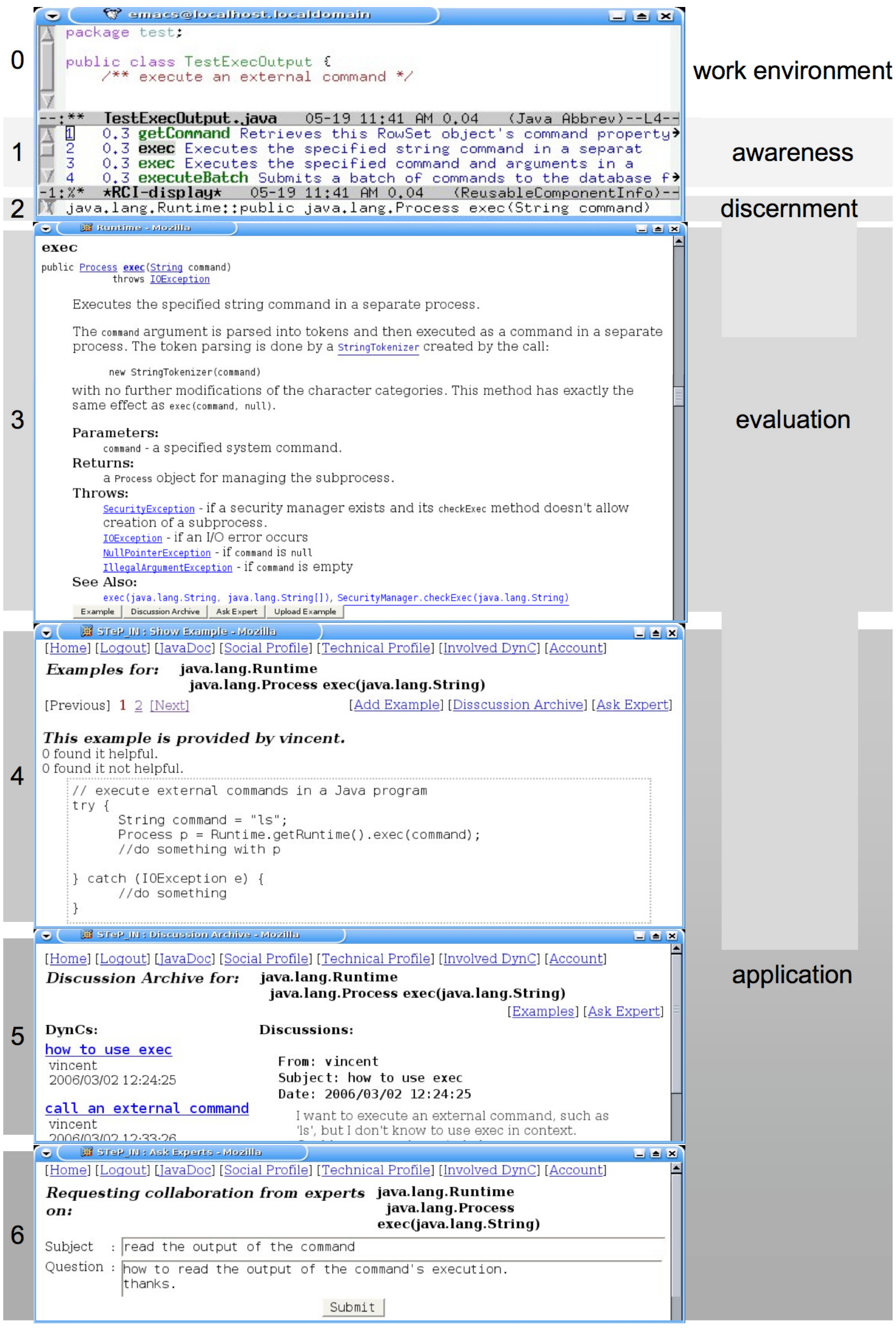
4.2.3 Jumping to the documentation

When the developer finds one promising reusable component and wants to know more about it, a click on the component name brings up a web browser that shows its full documentation (Figure 2–3) in the STeP_IN system, with four new buttons added to the standard Javadoc documents: `Example`, `Discussion Archive`, `Ask Expert`, and `Upload Example`. The document serves as the third layer of support to assist software developers in a detailed evaluation and application of the reusable component of interest. Some software developers who are not very familiar with the library may also need this layer of information to determine the task relevance of the component.

4.2.4 Finding examples

In some situations, a software developer might find that the document is inadequately written or does not explain the 'nitty-gritty' of applying the components. The developer can then press the `Example` button (Figure 2–3), which has been added to the standard Java documentation system in STeP_IN, to get example programs that illustrate how the component is used by other programmers (Figure 2–4). Those examples have been added previously by other developers through the `Upload Example` button. Examples serve as the fourth layer of support that helps the software developer understand the application of the component. This layer of information can also be used for evaluating the relevance of the component.

Figure 2 Layers of support for knowledge collaboration invoked on demand



```
package test;

public class TestExecOutput {
    /** execute an external command */

--:**  TestExecOutput.java   05-19 11:41 AM 0.04    (Java Abbrev)--L4--
1    0.3 getCommand Retrieves this RowSet object's command property
2    0.3 exec Executes the specified string command in a separat
3    0.3 exec Executes the specified command and arguments in a
4    0.3 executeBatch Submits a batch of commands to the database f
-1:%*  *RCI-display*   05-19 11:41 AM 0.04    (ReusableComponentInfo)--
java.lang.Runtime::public java.lang.Process exec(String command)
```

exec

```
public Process exec(String command)
             throws IOException
```

Executes the specified string command in a separate process.

The `command` argument is parsed into tokens and then executed as a command in a separate process. The token parsing is done by a `StringTokenizer` created by the call:

```
new StringTokenizer(command)
```

with no further modifications of the character categories. This method has exactly the same effect as `exec(command, null)`.

Parameters:
`command` - a specified system command.
Returns:
a `Process` object for managing the subprocess.
Throws:
`SecurityException` - if a security manager exists and its `checkExec` method doesn't allow creation of a subprocess.
`IOException` - if an I/O error occurs
`NullPointerException` - if `command` is `null`
`IllegalArgumentException` - if `command` is empty
See Also:
`exec(java.lang.String, java.lang.String[])`, `SecurityManager.checkExec(java.lang.String)`

[Home] [Logout] [JavaDoc] [Social Profile] [Technical Profile] [Involved DynC] [Account]

Examples for: **java.lang.Runtime**
java.lang.Process exec(java.lang.String)

[Previous] 1 2 [Next] [Add Example] [Disscussion Archive] [Ask Expert]

This example is provided by vincent.
0 found it helpful.
0 found it not helpful.

```
// execute external commands in a Java program
try {
      String command = "ls";
      Process p = Runtime.getRuntime().exec(command);
      //do something with p

} catch (IOException e) {
      //do something
}
```

[Home] [Logout] [JavaDoc] [Social Profile] [Technical Profile] [Involved DynC] [Account]

Discussion Archive for: **java.lang.Runtime**
java.lang.Process exec(java.lang.String)

[Examples] [Ask Expert]

DynCs:

how to use exec
vincent
2006/03/02 12:24:25

call an external command
vincent
2006/03/02 12:33:26

Discussions:

```
From: vincent
Subject: how to use exec
Date: 2006/03/02 12:24:25
```

I want to execute an external command, such as 'ls', but I don't know to use exec in context.

[Home] [Logout] [JavaDoc] [Social Profile] [Technical Profile] [Involved DynC] [Account]

Requesting collaboration from experts on: **java.lang.Runtime java.lang.Process exec(java.lang.String)**

Subject :	read the output of the command
Question :	how to read the output of the command's execution. thanks.

4.2.5 Reading the discussion archive

Examples may still not be enough for a software developer to understand how to apply the component in his or her development task. For example, a software developer might have concerns about issues related to performance or safety that are often neither discussed in documents nor illustrated in examples. The software developer can, by clicking on the `Discussion Archive` button, go to the archive of discussions that have been saved from previous knowledge exchanges among software developers about the component (Figure 2–5). The discussion archive is the fifth layer of support that complements the insufficiency of codified knowledge captured in formal documents and examples. The discussion archive assists in the evaluation phase, but the main purpose of the archive is to help software developers glean undocumented contextual information needed for applying the components in different real situations.

4.2.6 Finding experts for knowledge collaboration

If the developer's question does not have an existing answer in the discussion archive, the developer can click on the `Ask Expert` button to formulate a Dynamic Community (DynC) to enlist the help of peer developers through discussion. A DynC is a subgroup of knowledge workers that forms *ad hoc* in support of a particular *user* and a particular *task*, and disbands as the task is finished (Ye *et al.*, 2004). Unlike a static community, which forms around a particular domain and exists for a long time, a DynC forms for a particular task and exists only for a short period. The members of a DynC are selected by using the following two criteria:

1 They have expertise on the particular task.

2 They have established affinitive social contacts with the particular user.

The first criterion is grounded in the observation that in today's highly specialised world, expertise is no longer an absolute attribute of person but a relative function of a person *and* a task. In other words, experts can be identified only after the task is known. The second criterion is grounded in the findings that existing social contacts between the helpers and those they help could provide motivation to engage in knowledge collaboration based on the social norm of generalised reciprocity (Fischer *et al.*, 2004).

When a software developer asking for help clicks on the `Asking Expert` button, according to the principles of DynC, STeP_IN goes through two steps – expert identification and expert selection – to create the list of experts who should receive the help request. The expert identification process examines the relationship between the component and the experts to find all the members who have used the component before in the programs that they have written, and creates a list of *Candidate Helpers.* The expert selection process examines the social relationship from the asker to other developers extracted from their e-mail exchange history and previous interactions in the STeP_IN system to select from the list of *Candidate Helpers* a small group of people who have established a social relationship with the developer who is asking the question.

An e-mail is then sent to the *DynC Members* to ask for their help. The *DynC Members* who receive the e-mail requesting help can send back their help through e-mails, and the e-mails are captured by the system and stored in the database in association with the component. Such messages are then displayed for later members who click on the `Discussion Archive` button.

This is the sixth layer of support, which involves direct social interactions, and it is meant to be activated only when all the previous layers of technical support fail because social support requires the active participation of peers and incurs higher costs for knowledge collaboration.

4.2.7 Section summary

The integration of the two systems incrementally provides layered support for knowledge collaboration from awareness to social interaction, with a focus on the economical utilisation of human attention by accommodating the varying needs of each individual developer. Software developers fully control the provision of support; they can stop at any layer whenever they deem the presented information to be sufficient for their needs. For experts, the system remains brief and simple, without overloading information that takes too much attention from their real tasks; for novices, the system covers the complete spectrum of sociotechnical support, using human experts as the social infrastructure to back up the inadequacy that is inherent in knowledge repository systems due to the tacit nature of knowledge.

5 Conclusion

Focusing on the human and social aspects, this paper conceptualises software development as a knowledge-intensive and collaborative activity carried out by a group of developers with asymmetrically distributed knowledge. Knowledge collaboration with external knowledge repositories and knowledgeable peers is essential for software development. Knowledge collaboration in software development takes the forms of knowledge transfer and knowledge combination. The overall capability of a software project team is determined not only by the sum of the knowledge of individual developers, but also by the collaboration between developers and tools, and the collaboration among developers. Computational support for knowledge collaboration should become an integral part of software development environments. Computer systems can serve as equal partners that directly contribute knowledge to expand the knowing capability of developers, or they can serve as communication platforms that facilitate the knowledge transfer and combination among developers. Both approaches have merits and shortcomings and should be integrated to provide continuous support. Because individual developers have different background knowledge and development tasks, knowledge collaboration support has to be customised to the developer and his or her specific context. We propose the approach of layered support on demand that aims to accommodate the dynamically changing needs of each developer. This approach progressively provides more detailed external knowledge, depending on the situated knowledge needs of each developer, from awareness to full-fledged social interactions.

References

Ackerman, M.S. and Malone, T.W. (1990) 'Answer garden: a tool for growing organizational memory', *Proceedings of the ACM Conference on Office Information Systems*, Cambridge, Massachusetts, pp.31–39.

Becks, A., Reichling, T. and Wulf, V. (2004) 'Expertise finding: approaches to foster social capital', in M. Huysman and V. Wulf (Eds.) *Social Capital and Information Technology*, Cambridge, MA: The MIT Press, pp.333–354.

Berlin, L.M. (1993) 'Beyond program understanding: a look at programming expertise in industry', in C.R. Cook, J.C. Scholtz and J.C. Spohrer (Eds.) *Empirical Studies of Programmers: Fifth Workshop*, Palo Alto, CA: Ablex Publishing Corporation, pp.6–25.

Brooks, F.P.J. (1987) 'No silver bullet: essence and accidents of software engineering', *IEEE Computer*, Vol. 20, No. 4, pp.10–19.

Carey, T. and Rusli, M. (1995) 'Usage representations for reuse of design insights: a case study of access to online books', in J.M. Carroll (Ed.) *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: Wiley, pp.165–182.

Cross, R. and Borgatti, S.P. (2004) 'The ties that share: relational characteristics that facilitate information seeking', in M. Huysman and V. Wulf (Eds.) *Social Capital and Information Technology*, Cambridge, MA: The MIT Press, pp.137–161.

Detienne, F. (1995) 'Design strategies and knowledge in object-oriented programming: effects of expertise', *Human-Computer Interaction*, Vol. 10, Nos. 2–3, pp.129–169.

Fischer, G. (1987) 'A critic for LISP', *Proceedings of 10th International Joint Conference on Artificial Intelligence*, Los Altos, CA: Morgan Kaufmann, pp.177–184.

Fischer, G. (1999) 'Symmetry of ignorance, social creativity, and meta-design', *Knowledge-Based Systems*, Vol. 13, Nos. 7–8, pp.527–537.

Fischer, G. (2002) 'Desert island: software engineering – a human activity', *Automated Software Engineering*, Vol. 10, No. 2, pp.233–237.

Fischer, G. and Ostwald, J. (2001) 'Knowledge management – problems, promises, realities, and challenges', *IEEE Intelligent Systems*, January–February, pp.60–72.

Fischer, G., Scharff, E. and Ye, Y. (2004) 'Fostering social creativity by increasing social capital', in M. Huysman and V. Wulf (Eds.) *Social Capital and Information Technology*, Cambridge, MA: The MIT Press, pp.355–399.

Giddens, A. (1984) *The Constitution of Society: Outline of the Theory of Structure*, Berkeley: University of California Press.

Herbsleb, J. and Mockus, A. (2003) 'An empirical study of speed and communication in globally-distributed software development', *IEEE Transactions on Software Engineering*, Vol. 29, No. 3, pp.1–14.

Hollan, J., Hutchins, E. and Kirsch, D. (2001) 'Distributed cognition: toward a new foundation for human-computer interaction research', in J.M. Carroll (Ed.) *Human-Computer Interaction in the New Millennium*, New York: ACM Press, pp.75–94.

Hutchins, E. (1994) *Cognition in the Wild*, Cambridge, MA: The MIT Press.

Illich, I. (1971) *Deschooling Society*, New York: Harper and Row.

Lave, J. and Wenger, E. (1991) *Situated Learning: Legitimate Peripheral Participation*, Cambridge, UK: Cambridge University Press.

Lazzarato, M. (1996) 'Immaterial labour', in P. Virno and M. Hardt (Eds.) *Radical Thought in Italy: A Potential Politics*, Minneapolis: University of Minnesota Press, pp.133–147.

McDonald, D.W. and Ackerman, M.S. (1998) 'Just talk to me: a field study of expertise location', *Proceedings of Conference on Computer Supported Cooperative Work*, Seattle, Washington, pp.315–324.

Mockus, A. and Herbsleb, J. (2002) 'Expertise browser: a quantitative approach to identifying expertise', *Proceedings of 2002 International Conference on Software Engineering*, Orlando, Florida, pp.503–512.

Nahapiet, J. and Ghoshal, S. (1998) 'Social capital, intellectual capital, and the organizational advantage', *Academy of Management Review*, Vol. 23, pp.242–266.

Orlikowski, W.J. (2002) 'Knowing in practice: enacting a collective capability in distributed organizing', *Organization Science*, Vol. 13, No. 3, pp.249–273.

Polanyi, M. (1966) *The Tacit Dimension*, Garden City, NY: Doubleday.

Rajlich, V. and Gosavi, P. (2004) 'Incremental change in object-oriented programming', *IEEE Software*, Vol. 21, July–August, pp.62–69.

Robillard, M.P., Coelho, W. and Murphy, G.C. (2004) 'How effective developers investigate source code: an exploratory study', *IEEE Transactions on Software Engineering*, Vol. 30, No. 12, pp.889–903.

Schön, D.A. (1983) *The Reflective Practitioner: How Professionals Think in Action*, New York: Basic Books.

Suchman, L.A. (1987) *Plans and Situated Actions*, Cambridge, UK: Cambridge University Press.

Weinberg, G.M. (1971) *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold.

Ye, Y. and Fischer, G. (2002) 'Supporting reuse by delivering task-relevant and personalized information', *Proceedings of 2002 International Conference on Software Engineering*, Orlando, Florida, pp.513–523.

Ye, Y., Yamamoto, Y. and Kishida, K. (2004) 'Dynamic community: a new conceptual framework for supporting knowledge collaboration in software development', *Proceedings of 11th Asia-Pacific Software Engineering Conference*, Busan, Korea, pp.472–481.

Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y. and Asada, M. (2007) 'Searching the library and asking the peers: learning to use Java APIs on demand', *Proceedings of 2007 International Conference on Principles and Practices of Programming in Java*, Lisbon, Portugal: ACM Press (forthcoming).