Supporting Software Development as Knowledge-Intensive and Collaborative Activity

Yunwen Ye\textsuperscript{1,2}

\textsuperscript{1}SRA Key Technology Laboratory, 3-12 Yotsuya, Shinjuku, Tokyo 160-0004, Japan
\textsuperscript{2}Department of Computer Science, University of Colorado, Boulder, CO80309, USA
+1-303-492-3547
yunwen@cs.colorado.edu

ABSTRACT
Starting from the belief that software development is a human activity, this paper tries to conceptualize software development as a knowledge-intensive design and distributed cognitive activity. This conceptualization leads to the argument that providing support for software developers to engage in knowledge collaboration with external knowledge repositories and peers is essential for software development environments. Technical and social challenges in providing such support are identified, and an illustrative system support that we have been developing is briefly described.

Categories and Subject Descriptors

General Terms
Design, Human Factors, Theory.

Keywords
Knowledge collaboration, Distributed cognition, Software development support

1. INTRODUCTION
According to the IEEE Standard Glossary of Software Engineering Terminology, “software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is the application of engineering to software.” Engineering, as defined by the Oxford Dictionary of English, “is the branch of science and technology concerned with the design, building, and use of engines, machines, and structures.” As clearly articulated in the above definitions, the primary concerns of the research on software development have been the establishment of a quantifiable scientific foundation, toward the goal of reaching the stage where software systems can be built in a methodic approach through a specific and well-defined sequence of steps. By such doing, the productivity and quality of software development can be predicted and dictated mainly by the soundness of the approach rather than by the craft of individual developers.

Although software development do share many characteristics with many other engineering disciplines, software development presents new research challenges for us because it also heavily relies on the knowledge and creativity of individual software developers and their intense 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 human activity and focus on the cognitive 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 is more valued and encouraged. This viewpoint has been advocated by Weinberg [1], Brooks [2] and others [3], but it has not received enough attention in the main stream of software engineering research.

From this human activity perspective, our research focuses on the aspect of software development as a process of incremental crystallization of knowledge into the software systems, and then ponders upon its implications on the design of tools that support software developers. Seen in this light, software development is a knowledge-intensive and distributed cognitive activity [4] and it shares characteristics with other knowledge-intensive work such as design [5, 6]. Software development requires knowledge from several different domains, including both the computing domain and the application domain. 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 learning from and integrating the knowledge of various knowledge sources in the world. The development of software is therefore no longer confined to an individual developer but has to rely on distributed cognition by reaching into a networked world of information and computer mediated collaboration.

After introducing the basic theories of design and distributed cognition in Section 2, this paper tries to conceptualize software development as a knowledge-intensive design and distributed cognitive activity in Section 3, followed by Section 4 that argues for the needs of knowledge collaboration and the technical and social challenges therein. Section 4 illustrates approaches to overcome such challenges with a brief description of systems that we have been developing.
2. THEORETICAL FOUNDATIONS

This section describes theories from design and cognitive science that provide the basic conceptual framework of understanding the cognitive process of software development.

2.1 Software Development as Design Activity

Design is an activity of making representations to inform the process of making, in order to create artifacts we want to surround us [7]. Developing software is an instance of designing because the end product of software development is a representation (program) that can be executed algorithmically by computers, rather than an instance of making that ends up with physical artifacts. More fundamentally, software shares the most distinguished characteristic of other design domains: it requires an integrated response to a whole series of often conflicting issues through the integrating and combining a limited set of basic building blocks [6].

The 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, this design process, contrary to the beliefs of many software methodologists, does not follow any preset procedures [6, 8, 9]; it is dynamically determined by the knowledge that the designer has and the back-talk of the information presented in the workspace in which the designer is placed [10]. 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 [5]. Designers act to make representations, and simultaneously reflect upon the feedback presented by the representations, and then act further in an opportunistic way by constantly shifting attentions to critical parts of the problem and among several abstraction levels. The skills demonstrated by designers lies in how they are able to take situated actions [11] in response to the situational back-talk through reflection-in-action.

Because the information contained in the workspace and the representation are the major resource for reflection and then the following design actions, the important research questions for providing support for designers are: What, how and when extra information should be presented?

Conceptualizing software development as a design activity driven by reflection-in-action implies a new set of requirements for software development support environments: How can we increase the situational back-talk of developer actions? How can we enrich the development environment with more task relevant information and knowledge that stimulates reflection?

2.2 Software Development as Distributed Cognitive Activity

The notion of knowledge has expanded from its traditional focus on knowledge as expertise "stored as artifact" in a person’s head [12] to the focus on a person’s ability to find one’s way around the world, through effective interaction and collaboration with tools and people is the environment [13]. Following this strain of thinking, the distributed cognition theory broadens the units of analysis on cognitive systems. Traditional cognitive theories limit their analysis on an individual mind, focusing on the processes involved in a person’s head. Distributed cognition argues that to understand human intelligence in everyday activity we need to consider as a whole the large system of agencies that are assembled in different configurations and the role of tools and representations external to human minds [4]. In the real world, knowledge workers not only fully utilize the knowledge that they have learned in the past, but also exert their knowing capability—the ability of accessing and learning from external knowledge resources in the context of problem solving. Holland, Hutchins and Kirsch suggests that at least three kinds of distribution of cognitive process exist [4]:

1. Cognitive processes may be distributed across the members of a social group;
2. Cognitive processes may be distributed between internal and external (material or environmental) structure; and
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.

The third kind of distribution is in fact congruent with the reflection-in-action theory by Schon [5]. Earlier actions produce partial solutions that triggers reflection, which in turn leads to new actions that either deviate from previous plan or correct previous actions. Combined with the first kind of distributed cognition, we can further Schon’ theory on reflection-in-action. In Schon’s theory, action and reflection are both performed by a single designer. When the cognitive process is not limited to an individual, the back-talk of partial solutions created by one designer comes from not only his or her own reflection but also from the reflection of others. Furthermore, the subsequent actions triggered by the reflection can also be taken by other people.

Viewing software development in the light of distributed cognition, we can pose the following research challenges: What are the suitable configurations of developers, representations and tools that support software development better? How can we create appropriate socio-technical platforms in support of distributed cognition in software development?

3. KNOWLEDGE-INTENSIVE AND DISTRIBUTED SOFTWARE DEVELOPMENT

As software systems are getting more complicated and larger, the amount and kinds of knowledge required increase at a rapid speed. Even if the software itself might be a single one, because the computational environment in which it runs consists of large and complicated programs, it still demands substantially more knowledge than applications written in early days.

At the same time, modern software systems often contain tens of thousands of lines of code, and it has become hardly possible for a single developer to create such complicated and large systems; they have to be developed through the collaboration of many developers. Collaboration has become the essential part of software development, and distributed software engineering has emerged as an important research subfield [14].

However, most of the research in distributed software engineering 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 by the consequences of division of labor.
Another aspect of collaboration in software development that has not been paid enough attention is the brained power brought by multiple heads of developers, namely, the distributed intelligence brought by a group of developers that have mutually complementary knowledge.

With the current trend of globalization of companies, software companies are increasingly become distributed along different times zones, locations, and cultures. The distribution of software development has become necessary due to not only the needs of shifting labors to places that have lower costs, but also because of the pursuit of local talents that are otherwise unavailable. In other words, in addition to delegate the task of development to the most economically viable places, which is the current driving force of outsourcing, software development companies need also to ship the task to the most talented and suitable people regardless of location, time zone, and national boundary, which, we believe, will soon becoming the new driving force of outsourcing.

The success of many open source software projects offers an important lesson here. Open source software projects do not conform to the traditional transaction cost theory [15]; rather, they thrive on the particular capability for creating and sharing knowledge at a speed and efficiency superior to most software companies by distributing the work in alignment with the knowledge and skill owned by open source software developers because each developer picks up, by themselves, the task that they are mostly interested in and good at [16, 17].

4. KNOWLEDGE COLLABORATION IN SOFTWARE DEVELOPMENT

Viewing software development in the light of knowledge-intensive and distributed cognitive activity, we argue that an important aspect of supporting software development and software developers is to facilitate knowledge collaboration. Knowledge collaboration refers to the joint intellectual endeavor in which a knowledge worker engages with

   (1) external cognitive tools and
   (2) knowledgeable peers.

This ability of engaging in knowledge collaboration defines the knowing capability and competence of software developers.

4.1 Two Axes in Knowledge Collaboration

Knowledge collaboration can take place along two axes: the technological axis and the social axis.

Along the technological axis, knowledge workers enlist the help of external cognitive tools to complement the insufficient knowledge in their head. Cognitive tools include books, manuals and computerized information repositories such as on-line help systems, and repository systems. Our previous research has revealed the following cognitive challenges in utilizing external information and knowledge [18]:

- Software developers may not know the existence of useful external information and create suboptimal solutions;
- Software developers may not be able to find the right external information;
- Software developers may not be able to understand and apply the new information; and
- Different software developers have different knowledge needs.

When perusing the external information itself is not enough for learning, software developers often turn to knowledgeable peers for help, as one empirical study has found that one important factor that differentiates expert developers from novices is that experts are able to “use other experts faster” to help them solve problems collaboratively [19]. Hence, knowledge collaboration also takes place along the social axis by enlisting the help of other knowledgeable peers. The basic challenges in supporting knowledge collaboration along the social axis are:

- Software developers may not know to whom they can turn for help on this particular problem [20]; and
- Experts who are able to help may not be willing to due to the interruption to their own work and other various reasons [21].

4.2 Two Kinds of Computational Support

Because knowledge collaboration can take place along both technological and social axes, accordingly, computers can play two roles to support knowledge collaboration: as a cognitive tool to support software developers and as a mediating tool to provide a platform that facilitates knowledge collaboration among software developers.

4.2.1 Collaborating with Computers

Since the early days of computing, computer-based tools such as compilers, editors and reusable libraries have always been a quintessential part of the repertoire that consists of the technical competence of software developers. However, such tools have not become an equal partner in the underlying knowledge intensive process of software development. Software developers use such tools by delegating a part of development tasks such as compiling and linking, or by utilizing them to carry out a planned action such as editing. An equal partner in the knowledge collaboration process should directly contribute their own knowledge to the development process through augmenting the back-talk of user actions in order to complement the insufficient knowledge of software developers and to exert direct influences on subsequent actions of developers.

An example of research that view computers as equal partners in the knowledge collaboration process of software development is the LispCritic system [22], which is an intelligent agent that helps programmers improve their programming skills. It suggests better solution after it has recognized a less ideal code segment of programmers. Programmers can improve their skills by working together with the LispCritic system that provides feedback. The idea of developing intelligent agents to assist software development has been carried further in the development of the Argo design environment [23]. Argo is equipped with computer critics that deliver general software design knowledge for software developers to reflect upon their current design based on the recognition of insufficient knowledge demonstrated by software developers in their current development.

4.2.2 Computer-Mediated Knowledge Collaboration

In computer-mediated knowledge collaboration, computer plays the role of connecting software developers in need of new knowledge with developers who have the knowledge, instead of directly contributing to the knowledge creation process of software development. Computational environments provides a socio-technical platform for software developers to exchange and combine knowledge from different people. This line of research has been mainly conducted in CSCW (Computer-Supported Cooperative Work).

Because software development has traditionally been viewed as an individual cognitive process in software engineering, applying
CSCW research results to support software development has been few. The Answer Garden system [24] is an early effort in supporting Unix users to obtain help from other knowledgeable peers by routing questions asked by a user to pre-registered Unix experts. Terveen et al. developed a similar living design memory system [25] that connects software developers to share knowledge about software design. Expert Finder [26] is a system that directs a question posed by a Java developer to another one that has the matching expertise, so the expert developer can provide help through emails. Expertise Browser [27] mines CVS logs to find who have expertise on the portion of a system, so that the maintainer can direct his or her question to those experts. The Ariadne system [28] displays social call graphs that helps developers identify other developers affected by their actions and makes them aware of the needs for coordinated actions.

All of 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 are experts is only the first step, and the second step is those experts must be willing to offer help. To create computational platform to support knowledge collaboration, the motivation to participation has to be considered from the beginning, as Simon [10] has pointed out: “In any planning whose implementation involves a pattern of human behavior, that behavior must be motivated.”

4.3 The Need for Continuous Support

The above two roles that computers play in supporting knowledge collaboration are similar to the two threads of research in knowledge management: the repository-based approach and the community-based knowledge sharing approach.

Deeply influenced by research in artificial intelligence, the knowledge repository approach believes that expertise or knowledge can be externalized and formalized for sharing. The research focus, therefore, is on extracting knowledge by interviewing experts; formalizing extracted knowledge; storing formalized knowledge in knowledge repositories; and developing retrieval mechanisms to locate from repositories [29, 30].

Realizing the difficulties in the knowledge repository approach that cannot capture the tacit knowledge, a community-based approach was proposed [31]. 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 with the knowledge repository approach in which knowledge is managed, this approach advocates promoting informal communication in communities through the management of social network 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 completely at the control of the user who can enlist the collaboration of knowledge repository whenever he or she wants it, and the community-based approach is subject to the time and willingness of the other partner. The repository-based approach is more efficient, and the community-based approach requires user to invest time and efforts in advance to establish membership in the community. The major problem with the repository-based approach, as we have pointed out, is it cannot capture tacit and contextual knowledge.

An ideal situation is 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. By integrating these two approaches, continuous support can be provided for software developers to engage in knowledge collaboration depending on their unique needs.

5. ILLUSTRATIVE SYSTEM SUPPORT

Using one of the typical software development activity—software reuse—as an example to illustrate how to provide a continuous support for timely access to external information and expertise of knowledgeable peers for knowledge collaboration, this section briefly introduce two systems that we have been developing: CodeBroker [18] and STeP_IN (standing for Socio-Technical Platform for in situ Networking) [32]. The CodeBroker system addresses the cognitive challenges in knowledge collaboration along the technological axis (Section 4.1): being aware of the existence of external information (Java API library in this case) as well as being able to locate the needed information from a large repository; and the STeP_IN system addresses the challenges in the social axis of knowledge collaboration by helping a software developer find peer developers (1) who have the expertise on the information of interest and (2) who are likely willing to offer timely help. The STeP_IN system is based on a new conceptual framework Dynamic Community [33] that we have proposed to support situated knowledge collaboration. The integration of the two systems provides continuous support for knowledge collaboration from awareness to social support by conforming to the design principle of layered-presentation of information[34]...

5.1 Layered Presentation of Information

Successful knowledge collaboration requires software developers to locate, comprehend, and use external information in a timely fashion. This process involves at least three different phases: information discernment, detailed evaluation, and information application [35]. In the information discernment phase, users quickly determine whether a piece of information is relevant to their current task. The detailed evaluation phase involves a relatively thorough study of the information. In the information application phase, users need not only to understand the information but also to apply it in their current situation.

Each phase requires different levels of details of the information. For the information discernment phase, an overview of the information is enough. This phase also requires the simultaneous presentation of several candidates for users to compare and find the information that is most relevant. For the detailed evaluation phase, users need to focus on the details of a chosen piece of information. In the information application phase, examples that use the information are very effective in helping users understand, adapt, and integrate the information by providing context and enabling users to draw an analogy between their task and examples [36, 37].

The details required for information comprehension and use depend greatly on the knowledge of users. A continuum of demands for different levels of details exists. On one extreme, if the user already knows the presented information vaguely, an overview of the information may be enough. On the other extreme, if the user has never encountered the information before, he or she may need...
to go through all the three phases. Because of the tacit nature of knowledge [38], information repository cannot capture all the knowledge. In such cases, finding an expert on the topic becomes necessary for the user to understand and apply the information [27, 39].

The design principle of layered information-on-demand aims to provide a natural link from the task at hand to a variety of external information and knowledge resources, which are presented to users at different levels of abstraction in response to their different needs. Depending on the working context and background knowledge of a user, a unique network of information and knowledge is dynamically constructed and presented to provide a seamless transition from users’ interaction with tools to interaction with external information and other knowledgeable peers.

5.2 Continuous Support for Knowledge Collaboration

In the design of CodeBroker and STeP_IN that combine to become an integrated socio-technical platform in support of knowledge collaboration in software development, we have followed the principle of layered information-on-demand to focus on the economical utilization of human attention by presenting information in different levels of details and on demand from raising awareness to information overviews to details to examples and to experts. This design principle accommodate the differing needs of each individual user by putting control at the hand of users, who can stop at any level whenever they deem the presented information enough. For experts, the system remains brief and simple, without taking too much attention from their real task; for novices, the system has a complete socio-technical 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.

The following subsections describe the layers of information that are provided in the two systems.

Figure 1: The CodeBroker System

5.2.1 Autonomous Delivery to Raise the Awareness

In CodeBroker, software developers do not need to explicitly input queries for external information. As software developers enter document comments and method signatures in the editing space (Fig. 1(a)), the system extracts the comments and signatures and uses them as a query. A list of task-relevant components that match the query is autonomously retrieved from the Java API library and are delivered into the delivery buffer (Fig. 1(b)). This autonomous delivery makes it possible for software developers to use information that they do not even believe exists in the information repository, a well-recognized barrier to the utilization of information repository [18].

5.2.2 Personalized Information Overview

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

Each delivered component (Fig. 1(b)) has the rank of the relevance between the component and the current task, the relevance value, the name, and the synopsis of its functionality. This is a context-aware list of reusable components for the developer to browse and serves as the first layer of information presented on demand.

5.2.3 Mouse Movement-Triggered Information

Another important piece of information for a Java API method is its signature. We decide not to show it in the delivery buffer because signatures are usually long and take too much of the scarce screen resource. Furthermore, signature becomes relevant only when the developer finds the reusable component has the potential to be reused. In this sense, the signature of a component is secondary information needed to determine the reusability of the component. Therefore, only when the developer moves the mouse pointer over the component names in the delivery buffer (Fig. 1(b)), which triggers the display of the component’s full signature in the mini-buffer (Fig. 1(c)), the second layer of information is presented on demand to assist the developer in determining further the relevance of the component.

Figure 2: The STeP_IN System

5.2.4 Jumping To the Documentation

When the developer finds one promising reusable component and wants to know more about the component, a click on the component name brings up a web browser that shows its full documentation (Fig. 2) in the STeP_IN system (the STeP_IN system itself is an independent one that has its own search interfaces, for details, see [32]), with three new buttons added to the standard Java documentation.
5.2.5 Finding Examples
If the developer still does not know how to use the component, an example of its use might be helpful. He or she can then press the Examples button (Fig. 2), which has been added to the standard Java documentation system in STeP_IN, to get an example program that illustrates the use of the component.

5.2.6 Reading the Discussion Archive
When an example is still not enough for understanding or using the component, the developer can, by pressing the Discussion Archive button, go to the archive of previous discussions that are saved from previous information exchange about the component between helpers and helpees.

5.2.7 Finding Experts for Knowledge Collaboration
If the developer’s question does not have an existing answer in the discussion archive, he or she can press the Ask Experts button to formulate a dynamic community (DynC) for knowledge collaboration.

A DynC is a subgroup of knowledge workers that forms ad hoc in support of a particular user and a particular task, and assembles as the task is finished. Unlike a static community that forms around a particular domain and exists for a long time, a dynamic community forms for a particular task, and exists only for a short period. The members of a DynC are selected using the following two criteria:

1. They have expertise on the particular task, and
2. They already have social contacts with the particular user.

The first criterion is grounded in the observation that in today’s highly specialized world expertise is no more 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 assumption that existing social contacts between the helpers and the helpee could provide motivations to engagement in knowledge collaboration based on the social norm of generalized reciprocity [40].

When a member (i.e. a helpee) clicks the Asking Experts 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 relation from the component to the experts to find all the members who have used the component before in their programs, and create a list of Candidate Helpers.

The expert selection process examines the social relation from the helpee to other members extracted from their email exchange history and previous interactions in the STeP_IN system to select from the list of Candidate Helpers a small group of people that have established social relationship with the member.

An email is then sent to DynC Members to ask for their help. The DynC Members who receive the email requesting for help can send back their help through emails and the emails 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 the Discussion Archive button.

6. SUMMARY
This paper argues for the need for supporting knowledge collaboration in software development environments, based on the conceptualization of software development as knowledge-intensive and distributed cognitive activity. The technical and social challenges for supporting successful knowledge collaboration are described, and the approach to overcome such challenges is illustrated in the integration of two research prototypes: the CodeBroker and STeP_IN systems. The former creates the awareness of and the access to unknown reusable components, and the latter creates and facilitates collaboration among software developers as a social backup to complement the insufficiency of reuse repositories in support of software reuse.

ACKNOWLEDGMENTS
This research is supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan, Grant 15103 of the Open Competition for the Development of Innovative Technology program.

REFERENCES


