Chapter 11
Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments

Kumiyo Nakakoji, Yunwen Ye, and Yasuhiro Yamamoto

Abstract Looking at software development as a collective knowledge activity has changed the view of the role of communication in software development from something to be eliminated to something to be nurtured. Developer-centered collaborative software development environments (CSDEs) should facilitate software development in such a way, as individual software developers collaboratively develop information artifacts through social interactions. In this chapter, we identify two distinctive types of communication in software development, *coordination communication* and *expertise communication*, and argue that different sets of design guidelines are necessary in supporting each type of communication. We then describe nine design guidelines to support *expertise communication* based on the theories of social capital and models of supporting collective creativity.

11.1 Introduction

Software development is in essence information-intensive collaborative knowledge activity. It is about using information, generating information, and making information artifacts. The wide acceptance of agile processes and the success of many open source projects provide strong evidence that human aspects do matter in software development; cognitive and social processes play essential roles in successful software projects in which individuals' creative thinking in using and generating information are nurtured. We argue that software engineering environments must be designed to foster such individuals' creative knowledge processes, and that collaboration must be supported in the context of individuals' development activities. Collaborative software development environments (CSDEs) should be designed to facilitate and nurture individuals' creative knowledge processes. We call this approach *developer-centered CSDEs*.

K. Nakakoji (✉)
Research Centre for Advanced Science and Technology, University of Tokyo, Japan;
SRA Key Technology Laboratory Inc, Japan
e-mail: kumiyo@kid.rcast.u-tokyo.ac.jp

Collaboration takes place with or without explicit communication. On the one hand, software developers regularly engage in collaboration through artifacts without explicit communication (e.g., by writing comments in code to be read by others). On the other hand, explicit communication becomes necessary when developers must ask their peers for information that is otherwise not obtainable. Existing studies have provided ample evidence that both collocated and distributed software development teams frequently engage in communication to acquire necessary information from peer developers [24, 30, 32].

Such studies have made us aware that there are two distinctive types of situations in which developers communicate with their peers: one is when they want to coordinate development activities, and the other is when they want to acquire knowledge and understanding of a particular aspect of the software artifact under investigation. A developer engages in communication with peer developers in both situations by using the same communication channels (such as face-to-face, email, or chat), but the nature of the communication in each is quite different. Despite the quintessential differences in the nature of the goals, challenges, and concerns between these situations, studies on supporting communication in software development have not clearly separated the two.

We distinguish the two types of communication by calling the former *coordination communication* and the latter *expertise communication*, and argue that communication support must be tuned to each type of communication based on their inherent differences. Different sets of design guidelines need to be developed for supporting each type of communication in developer-centered CSDEs.

In this chapter, we first briefly describe the historical context for the developer-centered CSDE approach in software engineering research and discuss why communication must be supported as a first-class object in CSDEs. We then elaborate the differences between *coordination communication* and *expertise communication* and describe why different guidelines are necessary for supporting each type of communication. We finally present nine design guidelines for supporting *expertise communication*. We have derived these guidelines based on the theories of social capital [17] and models of supporting collective creativity [37, 38] as well as existing tools in the research fields of intelligent support, groupware, knowledge management, and organizational memory. We outline each guideline with theoretical grounds and illustrate each with technical instruments introduced by the existing tools and environments.

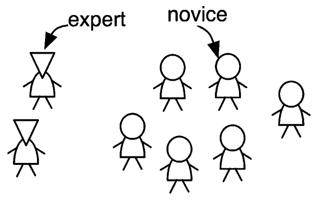
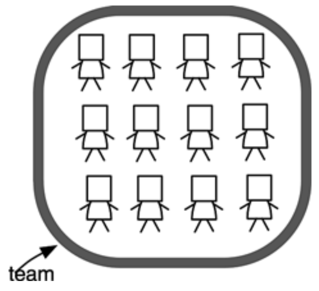
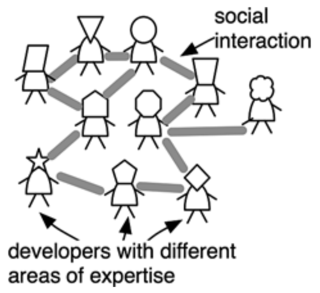
11.2 Historical Context: Three Schools of Research Toward Developer-Centered CSDEs

Software engineering research has looked at humans and their collaborations from its very beginning. During the last few decades, however, its emphases have shifted several times. We identify three distinctive schools of research in this particular area. Table 11.1 illustrates the differences among these three schools.

Table 11.1 Three schools of human-oriented software engineering research

	School 1 Psychology-centered	School 2 Process-centered	School 3 Developer-centered
Focuses on:	Human programming skills	Artifacts evolving processes	Cognitive and social processes
Software development is viewed:	As an individual skillful task	As an information processing task	As a creative knowledge task
Primary research concern:	How to develop programming skills	How to manage a project	How to nurture creative collavborative development processes
Developers interact with:	Programming language	Documents and artifacts through step-by-step actions	Tools and environments
Looks at:	Expert-novice differences	Team-performance difference	Individual developer difference in terms of areas of expertise
Related disciplines:	– Cognitive psychology – AI (Knowledge-based) – CHI	– Organizational science – Operation research – Measurements	– Tool building – Knowlwdge community – CSCW
Key phrases:	Psychology of programming	Process programming	Creativity, motivation, collaboration, community
Research materials:	– Performance and productivity	– Products (things) and processes (humans) – Project management	– Peer programers as knowledge resources – Making information relevant to the task-at-hand – Social dependencies among developers – Globally distributed collaborative development

Table 11.1 (continued)

	School 1 Psychology-centered	School 2 Process-centered	School 3 Developer-centered
Beliefs:	There ate two types of programmers: experts and non-experts (novices). Their interactions are of little relevance to the project performance. 	All programmers perform tasks more or less equally as information processors (like assembly lines) but the team as a whole performs differently depending on the organizational maturity. 	There are more than novice-expert differences among individuals. Each developer has his/her own area of expertise and are loosely tied together throught social interactions within a project.  social interaction developers with different areas of expertise

The first school of research, which we call the *psychology-centered* approach, has investigated the inner cognitive process of programming by focusing on the differences between expert and novice (non-expert) programmers through a number of psychological studies. That was the time right after the 1975 publication of Frederick Brook's *The Mythical Man-Months,* which basically says that the man-month is not an appropriate measure of software development project performance. It was realized that there is a huge performance difference between good programmers and not-so-good programmers. This had motivated a large number of studies to explore what psychological/cognitive factors in programming distinguish experts from novices. The psychology of programming is a research area that primarily looks at the differences of programming productivity and efficiency between experts and novices, while studying the benefits as well as difficulties of mastering programming features (e.g., the *if* statement design), methods (e.g., object orientation), and usage (e.g., mnemonic variable names) [48, 49].

The second school of research, which we call the *process-centered* approach, has its focus on the collaborative and managerial aspects of a software development project. It views software development as a group activity, or teamwork, and studies how to improve the capability of a software development organization, such as process traceability and repeatability [26]. Interestingly, this second school of research is less concerned with the programming skills of individual developers. Instead, it focuses more on the skills of organization. This school advocates that a software development process is programmable, and software development should be treated as assembly lines in which developers produce software by following predefined process instructions [39]. Developers take specification documents and then test specifications as input and produce source code and test cases as output. Researchers in this second school have primarily focused on how to help project management in orchestrating and coordinating a number of work pieces that have been produced by a large number of developers.

The third school of research, which we call the *developer-centered* approach, is the focus of our research. It looks at both the cognitive and social aspects of software development as well as their mutual interactions. The focus has returned to an individual developer, who is now viewed as having his or her own area of expertise in terms of a specific context, such as, the expertise on a piece of source code, the expertise on a certain feature of the program, the expertise on a certain aspect of the application domain, or the expertise on a certain programming language. Thus, symmetry of ignorance, or asymmetry of knowledge, exists among project members. They often have to collaborate with peer developers to accomplish their own programming tasks, and the success of the whole team depends on such collaborations.

Researchers in the third school explore how to support developers in such a way that they collectively develop information artifacts. Project managers are expected to be concerned with how to ensure the creativity and productivity of individual developers by providing physically, organizationally, culturally, and computationally *right* environments, rather than to worry only about how to quantify project performances and how to keep an eye on the project milestones with regard to the produced artifacts.

Two major factors have fueled the third school of research: open source communities and agile development methods. Both demonstrate the great importance of an individual developer's motivation, engagement, and communication in software development.

Since a large number of open source software development projects have emerged – making openly available their source code, related documents, development history data, and mailing list archives – a number of field studies have examined how software artifacts evolve through intensive communicative activities. As Augustin et al. who operated SourceForge, noted, such data have revealed that successful open source community projects "employed a number of practices that were not well characterized by traditional software engineering methodologies" [4]. Their paper lists mobility of resources, culture of sharing, and peer review and peer glory as examples of such practices, and labels the practices as "collaborative software development, or CSD."

Many of the twelve practices of XP [5], a representative agile method, are concerned with human and social aspects. By embracing individuals and interactions over processes and tools in their manifesto, agile software development methods aim to achieve successful software development by nurturing developer's collective creative processes [52].

Communication has long been regarded as an important activity in software development. A software engineering textbook published in 1985 by Fairley, for instance, shows that 37% of developers' time is spent in job communication and email [16]. However, communication was then regarded as an overhead rather than a part of the fundamental activities in software development. The trend of open source and agile methods has strongly hinted that communication needs to be treated as a first-class activity to be supported. The third school of research now views communication as something to be nurtured, not to be avoided.

It is very important to note that communication costs in software development remain very expensive, even in the eyes of the third school of research. We argue that although supporting communication is important, encouraging more communication in general should not be the research goal. Communication problems are caused not only by the lack of communicative acts, but sometimes by too many communicative acts. For example, one case study reported that overwhelming incoming mail messages resulted in a significant coordination problem [11]. Studies have shown that programmers in general prefer to work in a solitary environment with long periods of uninterrupted time during which they can concentrate [13]. By engaging in creative knowledge work, developers embrace flow experience, which is a situation "in which attention can be freely invested to achieve a person's goals, because there is no disorder to straighten out, no threat for the self to defend against" [10].

A developer-centered CSDE should first ensure that a developer can focus on his or her own task itself, and then facilitate easy communication with peer developers only when it becomes necessary. An important and often overlooked aspect is that when a developer wants to have communication, the person who is the recipient of this communication is also a developer. Supporting communication must carefully balance one developer's needs for communication and the other developer's needs for a concentrated flow experience.

11.3 Coordination Communication and Expertise Communication in Software Development

Many studies have observed how and about what developers communicate with one another during software development. For instance, through a study on three well-known open source projects, Gutwin et al. have found that text-based communications (mailing lists and chat systems) are the developers' primary sources of acquiring both general awareness of the entire team and more detailed information about people's expertise and activities [21]. In an ethnographic study on an industrial project, Ko et al. have analyzed what information needs developers face during software development [30]. The findings of this study indicate that coworkers were the most frequent source of information for software developers, and they were most frequently sought for the questions, "What have my coworkers been doing?" and "In what situations does this failure occur?"

Such studies demonstrate that two distinctive types of communication are involved in software development. One is what we call *coordination communication*, in which a developer communicates with his or her peers to discuss and negotiate in order to resolve conflicts or to avoid possible conflicts among the software components on which they are working. The structural dependency of software components may reflect "social dependency" among the developers who work on the components in the sense that they have to coordinate their tasks through social interactions when it is necessary to resolve perceived conflicts [28, 56]. Tools for supporting coordination communication have been primarily studied in such research areas as coordinating programmers and programming tasks, through making developers aware of what other developers are doing; for instance, Ariadne [14], Palantir [47] or FastDASH [6].

The other type of communication is what we call *expertise communication*, in which a developer communicates with his or her peers to ask for information that is essential for performing his or her own task at hand [32, 33, 58]. This is usually for obtaining knowledge and understanding about the design and/or behavior of a particular part of the system under development. Tools for supporting expertise communication have been primarily studied in such research areas as knowledge sharing and expert finding, helping developers ask questions of other developers; for instance, Expertise Recommender [34], Expert Browser [35] and STeP_IN [58].

The rather obvious separation of the two research areas reflects the fact that these two types of communication have quintessential differences in nature: in their goals, challenges and concerns. However, existing studies have not clearly separated and compared the two types of communication in designing communication support for CSDEs. One of the reasons for this might have been the fact that developers engage in both types of communication through the same communication channels: by sending email messages, by starting a chat, or by walking to a coworker's desk. However, different types of computational support mechanisms are necessary for the two types of communication due to their different natures.

For instance, a mechanism to find communication partners must be different in coordination communication and expertise communication because the relation between the developer who starts the communication and those with whom he or she communicates is different. In coordination communication, there is a symmetric or reciprocal relation between those who initiate communication and those who are sought for communication, with roughly equal amounts of interest and expected benefit. Coordination communication is a part of impact management, which is "the work performed by software developers to minimize the impact of one's effort on others and at the same time, the impact of others into one's own effort" [15].

In contrast, expertise communication is characterized by an asymmetric and unidirectional relation between the one who asks a question and the one who is asked to help [58]. The benefit is primarily for the information-seeking developer, while the costs are primarily paid by the information-provider. Such costs include the cost of paying attention to the information request, that of stopping his or her own ongoing development task, that of composing an answer for the information-seeking developer while collecting relevant information when necessary, and that of then going back to the original task.

We argue that different types of communication demand different sets of guidelines in designing communication support in developer-centered CSDEs. Redmiles et al. presented the continuous coordination paradigm for supporting software development [42]. The paradigm contains four principles: (1) to have multiple perspectives on activities and information; (2) to have nonintrusive integration through synchronous messages or through the representation of links between different sites and artifacts; (3) to combine socio-technical factors by considering relations between artifacts and authorship so that distributed developers can infer important context information; and (4) to integrate formal configuration management and informal change notification via the use of visualizations embedded in integrated software development environments [42]. Part of this paradigm supports coordination communication, and some, but not all, of its principles may also apply to support expertise communication.

In the remainder of this chapter, we present design guidelines for supporting expertise communication in software development. By "expertise communication," we do not mean knowledge exchange or knowledge transfer in a general sense. We use the phrase to refer to activities of a software developer who seeks, from his or her peer software developers, information that is essential yet not readily available in existing artifacts to accomplish his or her task, right in the middle of software development. The developer communicates with coworkers and asks for information not for the sake of increasing general knowledge in the abstract but to perform his or her own immediate task.

11.4 Nine Design Guidelines for Supporting Expertise Communication

This section presents nine design guidelines for supporting expertise communication.

Guideline #1: Expertise communication must be seamlessly integrated with other development activities.
A need for expertise communication emerges during the development activity when a software developer finds his or her task in need of information that is available only through other developers. The developer must be able to acquire the necessary information in a timely fashion so that he or she can carry out the current task more effectively and productively in a fluid manner [57]. Communication with peer developers to seek expertise should be supported as a continuum of information search tasks from an information-seeking software developer's point of view. It needs to be integrated with the software development environment to minimize the cognitive cost of conscientiously switching to a different application that supports expertise communication.

Not many existing tools supporting expertise communication consider this guideline. One of few tools that follow this guideline is STeP_IN_Java [58]. STeP_IN_Java has the "Ask Expert" feature embedded within the Java document-browsing interface. Each Java method is accompanied with the "Ask Expert" button; by pressing the button, the user is connected to a message-composing interface to write a question about the Java method, which is then delivered to those developers who have expertise about the method. The system thus makes expertise communication a natural extension of browsing Java documents.

Guideline #2: Expertise communication mechanisms should be personalized and contextualized for the information-seeking developer.
Information seeking in software development is an in situ and highly individualized action. A developer's needs for acquiring information from his or her coworkers arise when he or she is dealing with a specific task in a development environment. Integration with the development environment provides the context of the problem with which a developer is dealing. Such a context should be utilized by an expertise communication mechanism to customize its support to the context and the background knowledge of the developer [12, 57].

Identification of experts should be tuned for who is looking for what. Expertise is not an absolute attribute but a relative attribute of a developer, and it changes over time. Answer Garden [2] is an early attempt to identify UNIX experts based on predefined expertise profiles. The Expertise Recommender system [34] mines configuration management logs to identify experts based on organizational relations to support software maintainers. The developmental histories of developers (such as activities recorded in Concurrent Versions System (CVS) repositories, mailing archives, and written programs) should be used to identify who has the needed expertise about a particular problem at the particular moment [35, 55]. Having temporal information of the socio-technical context allows the information-seeker to understand whether a developer has the expertise being sought, and how he or she has gained it. Such information is not only useful for identifying the expertise being sought, but also valuable for understanding the information-seeker's background so that the system can locate those who have mental models similar to those of the information-seeking developer [55].

Guideline #3: Expertise communication should be minimized when other types of information artifacts are available.
Resorting to peers as information resources involves not only the information-seeking developer but also those developers who are asked to provide information [27]. Expertise communication is therefore an expensive means to get a developer's work done. It should not be promoted as the first choice; rather, it should be avoided when code, documents, development history records, archived previous communications, and/or other artifacts that satisfy the information needs are available.

Two mechanisms have been explored to consider this guideline in existing research: (1) initially leading users to artifacts before providing the means of expertise communication; and (2) archiving communication results to avoid unnecessarily repeated communications.

One example is Answer Garden and Answer Garden 2 [1, 2] which first allow a user to browse a database of commonly asked questions; if the sought answer is not present, the system "automatically sends the question to the appropriate expert, and the answer is returned to the user as well as inserted into the branching network, thus evolving the organizational memory [1]."

STeP_IN_Java [58] takes a similar approach by first guiding a developer in attending to the search and browsing interface of Java source code, documents, and communication archives. Only from the browsing interface does the system allow the developer to compose a question and ask other developers for information about the browsed artifact. The communication is again archived and associated with the artifact.

Other mechanisms, such as TagSEA, which is a shared waypoints mechanism to mark specific locations in Java source code elements or documents by using social tagging [50], are also useful in guiding developers to access previously communicated information.

Guideline #4: Expertise communication mechanisms should take into account the balance between the cost and benefit of an information-seeking developer and the group productivity.
From the project team's perspective, expertise communication is a two-edged sword in solving collaboration problems in software development. Broadcasting a question allows a developer to find the right people by letting other developers decide for themselves whether to respond [21]. However, if developers are frequently interrupted to offer help, their productivity is significantly reduced, resulting in lower group productivity for them [59].

Attention has been rapidly becoming the scarcest resource in our society [20]. Attention economy is concerned with the use or the patterns of allocation of attention for the best possible benefits. Following this thread of thought, the concept of collective attention economy has been proposed and used as an instrument to analyze the effective use of the sum of the attentions of the members in a group [59].

Our rough estimate of how much attention (in terms of time) is collectively spent in expertise communication in the mailing list of the open source project Lucerne is that more than 60,000 min (more than 1,000 h) were collectively spent every month [59]. In an organizational setting, this collective cost might even outweigh the benefits of knowledge collaboration; it certainly decreases the overall productivity of the whole project [41].

Some studies have looked into this problem. Both the Answer Garden approach [2] and the STeP_IN approach [58] try to reduce the cost incurred by expertise providers by limiting the recipients of the question only to those who are both able (through the expert identification process) and very likely to be willing (through the expert selection process) to answer the question.

Guideline #5: Expertise communication support mechanisms should consider social and organizational relationships when selecting developers for communication. Favorable interpersonal relationships help in communicating expertise due to pre-existing trust and mutual understanding [1]. An arduous relationship between an information seeker and an information provider often leads to the failure of expertise sharing [9]. People have very nuanced preferences concerning how and with whom they like to share expertise and how they like to maintain control of their social interactions [22].

The theory of social capital provides an analytic framework to understand this decision-making process [17]. 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" [36]. Social capital manifests itself in forms of obligations, expectations, trust, norms of generalized reciprocity, and reputations.

The feelings of expectation and obligation play important roles during the process of deciding whether and when to help. Researchers see obligations and expectations as complementary features [8] incurred during prior interactions that create value for the community in the future [44]. In other words, when B helps A, B would have a reasonable expectation that A will do something for B sometime down the road, and that A would feel obliged to help B [8].

Answer Garden 2 [1] uses organizational and physical proximities in the selection process. STeP_IN [58] uses social relationships and nuanced perception of individual relationships. Table 11.2 illustrates the different strategies used in the selection steps.

Similar to STeP_IN, some tools give high priority to the individual preferences for expertise communication. For instance, ReachOut [45] takes into consideration factors such as the helper's motivation to answer questions on the topic or

Table 11.2 Selection strategies reported in Answer Garden 2 [1], STeP_IN [58] and other strategies

Answer Garden 2 strategy	STeP_IN strategy	Other strategies
1. Organizational criteria 1-1 Keeping it local 1-2 Cross department 1-3 Last resort 2. Load on the sources 2-1 Selection based on regular workload 2-2 Selection based on workload over time 3. Performance 3-1 Problem comprehension 3-2 Providing a suitable explanation 3-3 Attitude	1. Inter-personal preferences of an individual 1-1 Exclude 1-2 Include 2. Obligation 2-1 Inter-personal obligation (has been helped by the information seeking developer) 2-2 Total-social obligation (has been helped by others in the group) 3. External communication history (has previously communicated via email) 4. Random selection	– Communication regency – Organizational hierarchy (relative significance and impact of the information-seeking developer to potential helpers) – Institutional secrecy – Eager helper (very motivated to help others) [54]

to participate at this very moment, as well as the helper's history of participation. The availability of choices and options helps the development of favorable attitudes toward expertise communication [46] and this favorable attitude is critical for expertise communication.

Guideline #6: Expertise communication support mechanisms should minimize the interruption when approaching those who are selected for communication.
When being approached to provide information for the benefit of another developer, developers are likely to feel interrupted. Answering or providing help consumes the time and attention of the helping developers and distracts them from their own tasks.

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" [51]. The cost of interruption includes not only the attention spent on the interrupting event, but also the disruption of flow and continuity of the ongoing work [29] and the accompanied work resumption efforts [28].

Expertise communication support tools, therefore, need to feature mechanisms that would minimize interruption when approaching potential helping developers. ReachOut [45], for instance, a chat-based tool for peer support, collaboration, and community building, invites potential helpers to join a conference chat by pushing the question to a nonintrusive client on their computer screens. Incoming questions fade in and out until the user decides to answer.

The field of human-computer interaction has long been studying how to model interruption between humans and computer agents [25]. Some parts of the models and findings of such studies should be taken into account to achieve more effective, less disruptive communication channels in support of expertise communication in software development.

In an attempt to minimize interruption for other developers by reducing the number of those who are asked to help, one may not be able to get the needed information. To address this issue, Answer Garden 2 has proposed the idea of escalation of support [1]. 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 involve more people, larger groups, and a wider range of areas.

Guideline #7: Expertise communication support mechanisms should provide ways to make it easier for developers to ask for help.
Developers feel different levels of difficulty and ease, depending on to whom they ask and through what communication channels. It is easy for developers to ask peers for information through face-to-face communication because they know each other, know how to approach each other, and have a good sense of how important their question is in relation to what the experts seem to be doing at the moment [23].

As Gerstberger and Allen report, “engineers, in selecting among information channels, act in a manner which is intended not to maximize gain, but rather to minimize loss. The loss to be minimized is the cost in terms of effort” [19]. Thus, developers tend to choose face-to-face communication because it would be less likely to be turned down, and to ask for help from coworkers whom they feel are easy to access rather than from the most appropriate person in some cases. This might end up in the wasteful use of a small set of “nice” people who keep helping others even if they do not have the appropriate expertise.

Developers may immediately get the necessary information or may never get any useful information, depending on how they ask. Rhetorical strategies, linguistic complexity, and wording choice all influence the likelihood of others responding [31] and replying to a question [3, 9].

Studies show that information-seekers 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 [9]. If every question asked would always go to all members of the mailing list, the information-seeker would risk giving colleagues the impression that he or she is rather ignorant and incompetent [18].

The perceived social burden on a potential information-provider may affect how easy it is for an information-seeker to ask a question. A field study of Answer Garden reports that because the information-seeker’s identity was not revealed in Answer Garden, the information-seeker felt less pressure in asking questions and bothering experts [2]. It might also become easier for an information-seeking developer to ask a question when he or she knows that the recipients have the option and freedom to ignore the request.

Reder and Shwab have noted that tactical skill in selecting communication channels “often determines an individual’s ability to influence and sometimes control the course and direction of group tasks and impact the success of particular projects” [41]. Expertise communication support mechanisms, therefore, need to consider social factors that affect expertise-seeking behaviors and help software developers in their expertise communication if they do not have the tactical skill to select the right communication channel.

Guideline #8: Expertise communication support mechanisms should provide ways to make it easier for developers to answer or not to answer the information request. Developers who receive the request for help in expertise communication need to decide whether to answer. They may feel different levels of social pressure, depending on from whom and through which communication channel the request is coming. For instance, in direct emails, the receiver bears the interruption cost of the reply or the social burden of taking no action [53].

The success of expertise communication should not come at the price of developers' reluctance for further participation in future collaboration. Some developers might get bored by answering repeatedly asked questions that they deem too simple to be worth their time and expertise, and some might want to guard their unique expertise to retain their "market value" in the organization [43]. The goodwill and limited attention of developers should be economically utilized to achieve sustainable and long-term success. They should not be forced into helping just for fear of causing unnecessary disruptions to the social cohesion and norms of the project team, which is unlikely to be sustainable.

Unwillingness also leads to lower quality of communication. When workers are forced into sharing expertise without much willingness, they often use "verbal and intellectual skills as a defense to keep a person with a problem from consuming too much of their time," and their answers are often "impressive-sounding" but not helpful [9] resulting in a waste of time for both parties.

Developers 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 may incur a social cost. Saying "no" untactful to an information-seeking developer deteriorates the expert's relation with the seeker and negatively affects the expert's social reputation among other peers because such behavior deviates from social norms [40].

The STeP_IN framework provides a communication mechanism called a *dynamic mailing list*; a temporal mailing list is created every time an information-seeking developer posts a question, with the recipients decided dynamically [58]. Whereas the sender's identity is shown to the recipients, the recipients' identities are not revealed unless they reply to the request. If some of the recipients do not answer, for whatever reasons, nobody will know it; therefore, refusing to help becomes socially acceptable, similar to "hiding out to get some work done" [13]. If one of the recipients answers the question, his or her identity is revealed to all members of the dynamic mailing list. This asymmetrical information disclosure is meant to reinforce positive social behaviors without forcing others into collaboration.

Guideline #9: Expertise communication channels must be socially aware.
Socially aware communication [40] refers to the transmission of information or signals that does not violate social norms. Existing communication channels include

face-to-face, direct email, mailing lists, wikis, bulletin boards, Internet relay chat (IRC), telephone, or video conferences.

Different communication channels give various degrees of control to either the information-seeking developer or those who are asked to provide information. Decisions need to be made, depending on the goals and social context, about who should gain the social control of communication.

One prime example of such control is the disclosure of identities of information-seekers and information-providers. Different tools take different approaches in designing such disclosure of identities. In a field study of Answer Garden that had an information-seeker's identity hidden and an information-provider's identity revealed, the seekers felt easier asking and the information-providers felt more "obliged" and tended to "show off" their expertise [2]. STeP_IN [58] in contrast, makes a seeker's identity revealed to those who receive the question, whereas the receivers' identities remain hidden unless they answer in a dynamically formulated temporal mailing list. This design decision is based on the viewpoint that the information-provider should be granted more control because the information-seeker is the main beneficiary and the information-provider is the benefactor.

Cohen et al. have investigated, through field studies of a legal firm, the phenomena of adversarial collaboration, in which peers who are adversaries having opposing goals nonetheless have to collaborate to get their tasks done [7]. They argue that adversarial collaborations are "the sine qua non of situations that call for the selective dissemination of information." Although software developers in a project are by no means adversaries and have no opposing goals, they may have different interests and motivations in their own specific contexts, especially when a project is inter-organizational or involves subcontracted members. Mechanisms for supporting asymmetric disclosure of information may need to be designed within expertise communication channels.

11.5 Concluding Remarks

This chapter has argued for a developer-centered CSDE where communication is considered as a first-class activity in software development. We identified two distinctive types of communication in software development, *coordination communication* and *expertise communication*, and elaborated on their differences.

Communication support mechanisms have features that imply suitable communication genres [41]. Such features include whether the communication is one-to-one or one-to-many; whether the communication happens synchronously or asynchronously; whether the sender and the recipients are anonymous or identified; whether all the relevant information is disclosed symmetrically or asymmetrically among the sender, recipients, and others; whether the social control of communication is granted to the sender or to the recipient; whether the mechanism makes it easier for the information-seeker or the recipient; and what media should be used, such as text, voice, video, or other types of multimedia, each of which demonstrates different degrees of achievability and searchability.

Taking the above features into total consideration as well as the distinctive nature of expertise communication in software development, we have presented a list of nine design guidelines for supporting expertise communication in software development. These guidelines are interdependent: following one guideline may also lead to following a few other guidelines, or following one guideline may conflict with following another guideline. Each guideline is important in some particular context. In designing expertise communication support mechanisms, one needs to understand what corporate and organizational culture exists and what types of collaboration their software projects want to nurture.

Although this chapter has argued to distinguish coordination communication from expertise communication for supporting communication in developer-centered CSDEs, it has not been our intention here to develop two different communication interfaces for developers. Developers presently do not and probably will not want to distinguish the two; they simply want to communicate with their peers for a variety of reasons. After identifying different sets of design guidelines in support of coordination and expertise communications, the forthcoming research agenda would involve how to integrate the two mechanisms so that developers would be able to seamlessly engage in different types of communications without consciously switching between the two.

References

1. Ackerman MS, McDonald DW (1996) Answer Garden 2: Merging organizational memory with collaborative help. Proceedings of CSCW'96, ACM Press, New York, pp. 97–105.
2. Ackerman MS (1998) Augmenting organizational memory: A field study of Answer Garden. ACM Transactions on Information Systems 16(3): 203–224.
3. Arguello J, Butler BS, Joyce E, Kraut R, Ling KS, Rose C, Wang X (2006) Talk to me: Foundations for successful individual-group interactions in online communities. In: Grinter R, Rodden T, Aoki P, Cutrell E, Jeffries R, Olson G (Eds.) Proceedings of CHI'06, April 22–27, ACM, New York, pp. 959–968.
4. Augustin L, Bressler D, Smith, G (2002) Accelerating software development through collaboration. Proceedings of ICSE'02, ACM, New York, pp. 559–563.
5. Beck K (1999) Extreme Programming Explained: Embrace Change. Reading, MA: Addison-Wesley.
6. Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: A visual dashboard for fostering awareness in software teams. Proceedings of CHI'07, ACM, New York, pp. 1313–1322.
7. Cohen AL, Cash D, Muller MJ (2000) Designing to support adversarial collaboration. Proceedings of CSCW'00, ACM, New York, pp. 31–39.
8. Coleman JC (1988) Social capital in the creation of human capital. American Journal of Sociology 94: S95–S120.
9. Cross R, Borgatti SP (2004) The ties that share: Relational characteristics that facilitate information seeking. In: Huysman M, Wulf V (Eds.) Social Capital and Information Technology. Cambridge, MA: The MIT Press, pp. 137–161.
10. Csikszentmihalyi M (1990) Flow: The Psychology of Optimal Experience. New York: HarperCollins.

11. Damian D, Izquierdo L, Singer J, Kwan I (2007) Awareness in the wild: Why communication breakdowns occur. Proceedings of ICGSE'07, IEEE Computer Society, Washington, DC, pp. 81–90.
12. Davor Cubranic C, Murphy GC (2003) Hipikat: Recommending pertinent software development artifacts. Proceedings of ICSE'03, Portland, OR, pp. 408–418.
13. DeMarco T, Lister T (1999) Peopleware: Productive Projects and Teams. New York: Dorset Housing Publishing.
14. de Souza CRB, Quirk S, Trainer E, Redmiles D (2007) Supporting collaborative software development through the visualization of socio-technical dependencies. Proceedings of GROUP'07, Sanibel Island, FL, pp. 147–156.
15. de Souza CRB, Redmiles D (2008) An empirical study of software developers management of dependencies and changes. Proceedings of ICSE'08, pp. 241–250.
16. Fairley R, (1985) Software Engineering Concepts. New York: McGraw-Hill College.
17. Fischer G, Scharff E, Ye Y (2004) Fostering social creativity by increasing social capital. In: Huysman M, Wulf V (Eds.) Social Capital and Information Technology. Cambridge, MA: The MIT Press, pp. 355–399.
18. Flammer A (1981) Towards a theory of question asking. Psychiatry Research 43: 407–420.
19. Gerstberger PG, Allen TJ (1968) Criteria used by research and development engineers in the selection of an information source. Journal of Applied Psychology 52(4): 272–279.
20. Goldhaber MH (1997) The attention economy. First Monday 2(4).
21. Gutwin C, Penner R, Schneider K (2004) Group awareness in distributed software development. Proceedings of CSCW'04, ACM, New York, pp. 72–81.
22. Halverson CA, Erickson T, Ackerman MS (2004) Behind the help desk: Evolution of a knowledge management system in a large organization. Proceedings of CSCW'04, ACM, New York, pp. 304–313.
23. Herbsleb J, Grinter RE (1999) Splitting the organization and integrating the code: Conway's law revisited. Proceedings of ICSE'99, pp. 85–95.
24. Herbsleb J, Mockus A (2003) An empirical study of speed and communication in globally-distributed software development, IEEE Trans Software Engineering 29(3): 1–14.
25. Horvitz E, Apacible J (2003) Learning and reasoning about interruption. Proceedings ICMI'03, ACM, New York, pp. 20–27.
26. Humphrey W (1989) Managing the Software Process. Reading, MA: Addison-Wesley Professional.
27. Illich I (1971) Deschooling Society. New York: Harper and Row.
28. Iqbal ST, Bailey BP (2006) Leveraging characteristics of task structure to predict the cost of interruption. CHI'06, ACM, New York, pp. 741–750.
29. Jackson T, Dawson R, Wilson D (2001) The cost of email interruption, Journal of Systems and Information Technology 5: 81–92.
30. Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. International Conference on Software Engineering (ICSE), 20–26 May, pp. 344–353.
31. Kraut R, Kiesler S, Mukhopadhya T, Scherlis W, Patterson M (1998) Social impact of the internet: What does it mean? Commun ACM 41(12): 21–22.
32. LaToza TD, Venolia G et al (2006) Maintaining mental models: A study of developer work habits. Proceedings of ICSE'06, Shanghai, pp. 492–501.
33. McDonald DW, Ackerman MS (1998) Just talk to me: A field study of expertise location. Proceedings of CSCW'98, Seattle, WA, pp. 315–324.
34. McDonald DW, Ackerman MS (2000) Expertise recommender: A flexible recommendation system architecture. Proceedings of CSCW'00, pp. 101–120.
35. Mockus A, Herbsleb J (2002) Expertise browser: A quantitative approach to identifying expertise. Proceedings of ICSE'02, Orlando, FL, pp. 503–512.
36. Nahapiet J, Ghoshal S (1998) Social capital, intellectual capital, and the organizational advantage. Academy of Management Review 23: 242–266.

37. Nakakoji K (2006) Supporting software development as collective creative knowledge work. Proceedings of KCSE2006, Tokyo, pp. 1–8.
38. Nakakoji K, Ohira M, Yamamoto Y (2000) Computational support for collective creativity. Knowledge-Based Systems Journal, Elsevier Science 13(7–8): 451–458.
39. Osterweil L (1987) Software processes are software too. Proceedings of ICSE'87, pp. 2–13.
40. Pentland A (2005) Socially aware computation and communication. Computer 38(3): 33–40.
41. Reder S, Schwab RG (1988) The communication economy of the workgroup: Multi-channel genres of communication. Proceedings of CSCW'88, ACM, New York, pp. 354–368.
42. Redmiles D, Hoek Avd, Al-Ani B, Hildenbrand T, Quirk S, Sarma A, Filho RSS, de Souza C, Trainer E (2007) Continuous coordination: A new paradigm to support globally distributed software development projects. Wirtschaftsinformatik 49: S28–S38.
43. Reichling T, Veith M (2005) Expertise sharing in a heterogeneous organizational environment. Proceedings of ECSCW'05, Springer-Verlag, New York, pp. 325–345.
44. Resnick P (2002) Beyond bowling together: Sociotechnical capital. In Carroll JM (Ed.) HCI in the New Millennium. Reading, MA: Addison-Wesley, pp. 247–272.
45. Ribak A, Jacovi M, Soroka V (2002) Ask before you search: Peer support and community building with Reach out. Proceedings of CSCW'02, ACM, New York, pp. 126–135.
46. Salancik GR, Pfeffer J (1978) A social information processing approach to job attitudes and task design. Administrative Science Quarterly 23: 224–253.
47. Sarma A, Noroozi Z, Hoek Avd (2003) Palantir: Raising awareness among configuration management workspaces. Proceedings of ICSE'03, pp. 444–454.
48. Shneiderman B (1980) Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop.
49. Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. IEEE Transactions on Software Engineering 10(5): 595–609.
50. Storey M, Cheng L, Bull I, Rigby P (2006) Shared waypoints and social tagging to support collaboration in software development. Proceedings of CSCW'06, ACM, New York, pp. 195–198.
51. Szoestek AM, Markopoulos, P (2006) Factors defining face-to-face interruptions in the office environment. Proceedings of CHI'06, ACM, New York, pp. 1379–1384.
52. Tomayko JE, Hazzan O (2004) Human Aspects of Software Engineering (Electrical and Computer Engineering Series). Rockland, MA: Charles River Media, Inc.
53. Tyler JR, Tang JC (2003) When can I expect an email response? A study of rhythms in email usage. Proceedings of ECSCW'03, Helsinki, pp. 239–258.
54. Van den Hooff B, De Ridder JA, Aukema EJ (2004) Exploring the eagerness to share knowledge: the role of social capital and ICT in knowledge sharing. In: Huysman M, Wulf V (Eds.) Social Capital and Information Technology. Cambridge, MA: The MIT Press, pp. 163–186.
55. Vivacqua A, Lieberman H (2000) Agents to assist in finding help. Proceedings of CHI'00, ACM, New York, pp. 65–72.
56. Wagstrom P, Herbsleb J (2006) Dependency forecasting. Communications of the ACM 49(10): 55–56.
57. Ye Y, Fischer, G (2002) Supporting reuse by delivering task-relevant and personalized information. Proceedings of ICSE'02, Orlando, FL, pp. 513–523.
58. Ye Y, Yamamoto Y, Nakakoji K (2007) A socio-technical framework for supporting programmers. Proceedings of ESEC/FSE'07, ACM, New York, pp. 351–360.
59. Ye Y, Yamamoto Y, Nakakoji K (2008) Understanding and improving collective attention economy for expertise sharing. Proceedings of CAiSE'08, June, Lecture Notes in Computer Science 5074, Springer, Berlin Heidelberg, pp. 167–181.