

# The Awareness Network, *To Whom* Should I Display My Actions? And, *Whose* Actions Should I Monitor?

Cleudson R. B. de Souza, David F. Redmiles, *Member, IEEE*

**Abstract**—The concept of awareness plays a pivotal role in research in Computer-Supported Cooperative Work. Recently, Software Engineering researchers interested in the collaborative nature of software development have explored the implications of this concept in the design of software development tools. A critical aspect of awareness is the associated coordinative work practices of displaying and monitoring actions. This aspect concerns how colleagues *monitor* one another's actions to understand how these actions impact their own work and how they *display* their actions in such a way that others can easily monitor them while doing their own work. In this paper, we focus on an additional aspect of awareness: the identification of the *social actors who should be monitored and the actors to whom their actions should be displayed*. We address this aspect by presenting software developers' work practices based on ethnographic data from three different software development teams. In addition, we illustrate how these work practices are influenced by different factors, including the organizational setting, the age of the project, and the software architecture. We discuss how our results are relevant for both CSCW and Software Engineering researchers.

**Index Terms**—Computer-supported cooperative work, Organizational management and coordination, Programming environments, Programming teams, Tools.

## 1 INTRODUCTION

SOFTWARE development, being a human activity, is challenged by human limitations. There are individual cognitive challenges and social collaborative challenges. The collaborative challenges are what we are concerned with in this work, having observed teams of software developers working together to deliver their target products.

Collaborative challenges were identified very early in the nascent field of Software Engineering. Brooks [5] observed that software development was “a complex interpersonal exercise.” The seminal work by Curtis and colleagues recognized that breakdowns in communication and coordination efforts constituted a major problem in large-scale software development [12]. Later, Staudenmayer [49] recognized that good coordination of teams of developers was correlated with high team performance. Finally, Herbsleb et al. [33] documented how software development tasks performed in distributed contexts took longer than similar tasks performed in collocated ones due to the cost of coordinating developers in different geographical locations. Over the years, Software Engineering practitioners have proposed a large number of strategies to facilitate the collaboration required of software development efforts including tools (e.g., CVS), ap-

proaches (e.g., software process models), and techniques (e.g., pair programming).

Many of the problems faced by software developers are the same as problems faced by professionals in other domains: communication breakdowns, coordination problems, lack of knowledge about colleagues' activities, and so on [46]. The research field of Computer-Supported Cooperative Work (CSCW) emerged to address these kinds of problems, and, in the beginning, primarily in the area of office automation. Generally, CSCW as a field concentrates on understanding how collaboration among individuals takes place, and how it can efficiently be supported by (computational) tools [25, 45].

Since its inception, CSCW has attracted researchers who have considered Software Engineering as an important domain for research [25]. Although there has always been some crossover, recently researchers in Software Engineering have become increasingly interested in the lessons from the CSCW literature. The timing could not be more critical due to various trends, including the growth of globally distributed projects that exacerbate coordination and communication problems and agile methods that emphasize cooperation and communication among software developers [17]. Among the examples of Software Engineering research informed by CSCW concepts is that by Sarma and colleagues [43], who argue that awareness among software developers involved in programming activities is important because configuration management workspaces create a harmful isolation among software developers. Estublier and Garcia [21] present a similar argument, but they discuss it in the con-

- C. R. B. de Souza is with IBM Brazil, São Paulo, SP, Brazil, 04007-005. Email: [cleudson.desouza@acm.org](mailto:cleudson.desouza@acm.org). While doing this work, he was at the Federal University of Pará, Belém, PA, Brazil, 66075-110.
- D. F. Redmiles is with the Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA, 92617. E-mail: [redmiles@ics.uci.edu](mailto:redmiles@ics.uci.edu).

Manuscript received (insert date of submission if desired).

text of software process models. Another example is our own previous work [16], wherein we discuss the different roles played by application programming interfaces (APIs) in the coordination of software developers' work. Finally, Cataldo and colleagues discuss the importance of work dependencies on failure prediction [8].

In the growing body of Software Engineering research influenced by CSCW, the concept of awareness has had considerable influence. For instance, this concept has influenced the design of several collaborative development environments, including Ariadne [13, 53]; CollabCVS [32]; FastDASH [4]; Jazz [10]; and Palantir [43]. More recently, Treude and Storey [54] investigated how awareness was achieved in a large-scale project through the usage of tools that aggregate information from different sources.

One of the most commonly cited descriptions of awareness is the one by Dourish and Bellotti: "awareness is an understanding of the activities of others, which provides a context for your own activity [18]." In addition, they argue that, "awareness information is always required to coordinate group activities, whatever the task domain." This fundamental nature of awareness and its growing use in collaborative software environments motivates us to examine it more carefully.

Intuitively, a question that arises from the concept of awareness is who are the others that one should be aware of and vice versa? Another question is in what ways is awareness achieved? We explore these questions more carefully in the next section, but, for now, we can introduce the focus of this paper as the identification of the *awareness network*—the network of actors whose actions need to be monitored by an actor and those to whom this actor needs to make his or her own actions visible. An actor displays his or her actions so that others can understand the impact of the work. Similarly, actors are monitored because their actions can impact the work of an actor. In short, awareness allows impact management, which is important for coordinating collaborative work [15]. Our main contributions in this paper are to illustrate how software developers identify and maintain their awareness networks, to describe features of these networks, and to discuss how different aspects of the work facilitate or hinder the identification and maintenance of awareness networks. In this regard, this paper illustrates how the software architecture influences the awareness networks.

The data used to ground the conclusions of this paper are based on three different software development teams: Alpha, Beta, and Gamma. The Alpha team was performing maintenance of its software, whereas the Beta and Gamma teams were implementing the first releases of their software. Furthermore, while our data collection and analysis in the Beta team focused solely on the implementation phase, our focus in the Gamma team was on the whole software development process from requirements to testing, including implementation. Therefore, our data sets were complementary strengthening our results.

The rest of this paper is organized as follows. The next section provides more background from the literature especially about why we focus on the concept of the

awareness network and its interrelationship with work practice. The section thereafter describes the three research sites studied, Alpha, Beta, and Gamma, as well as the methods used to collect and analyze data from these sites. Next, the ethnographic data of each team is presented illustrating how team members identified their awareness network as well as the aspects that influenced this identification. A discussion follows in the subsequent sections, including aspects of the awareness network identification and management, as well as aspects that influence this network. This discussion illustrates the relevance of the contributions presented in this paper for CSCW and Software Engineering researchers. Finally, the last section presents our final considerations.

## 2 BACKGROUND ON AWARENESS IN CSCW RESEARCH

Many, if not most, authors adopt the aforementioned description of awareness by Dourish and Bellotti [18]. However, examining awareness in more depth and based on previous research on CSCW quickly reveals the many different ways awareness has been used.

Schmidt [44] provides a brief but comprehensive review of over a dozen uses of the term in the literature and everyday life. He then connects awareness to work practice [27-29], and conceptualizes awareness as a range of coordinative practices performed by competent actors to accomplish their work [31]. These coordinative practices take place while actors are performing their work—that is, *awareness is an attribute of action, not an aspect separate from it*. Schmidt clearly summarizes this point: "doing one thing while taking heed of other relevant occurrences are not two parallel lines of action but a specific way of pursuing a line of action, namely to do it heedfully, competently, mindfully, accountably."

Furthermore, the nature of these coordinative practices is dual: it involves (i) *displaying* one's actions, and (ii) *monitoring* others' actions. That is to say, social actors *monitor* their colleagues' actions to understand how these actions impact their own work and, while doing their work, social actors also *display* their actions in such a way that others can easily monitor them, but without disrupting their colleagues.<sup>1</sup> *The displaying and the monitoring of activities are thus complementary aspects: the displaying of one's actions is facilitated by the monitoring of the others and vice versa*. As an example, in a seminal paper about awareness, Heath and Luff [29] describe how coordination takes place in the London subway control rooms. In this setting, one of the actors, the *DIA*, is responsible for keeping the public informed about the state of the subway service. The *Controller* is another actor who deals with the train drivers, giving them directions about when, where, and for how long they should stop the trains. The *DIA* listens the *Controller* to give directions to a train driver on the phone, and even before this conversation finishes, the *DIA* announces changes in the service to the

<sup>1</sup> Implicit in this discussion is the notion of interdependent activities, that is, displaying and monitoring are especially relevant because the outcome of one's action can affect others' actions [37, 44].

public. Similarly, the Controller will stress some words during his conversations with train drivers to emphasize changes in the service that the DIA needs to announce to the public. In short, the monitoring of the Controller by the DIA is facilitated by the display of actions that are performed by the Controller. Heath and Luff emphasize that these two aspects of work, the displaying and monitoring, were possessed by all Controllers and DIAs observed, and they were not specific to a certain subset of individuals.

According to Schmidt, these work practices often are not viewed by researchers as the result of deliberate, explicit actions; but he counters that this should not be the case because social actors deftly choose the degree of obtrusiveness of their actions [44]:

No clear distinction exists between, on the one hand, the coordinative practices of monitoring and displaying, normally referred to under the labels “mutual awareness” or “peripheral awareness,” and, on the other hand, the practices of directing attention or interfering for other purposes. In fact, by somehow displaying his or her actions, the actor is always, in some way and to some degree, intending some effect on the activities of colleagues. The distinction is not categorical but merely one of degrees and modes of obtrusiveness.

In other words, while awareness usually has been associated with actors’ achievements, this is not always true: checking a tool log, sending an email, or starting a conversation are all valid examples of work practices deftly used by social actors to become aware of their colleagues’ actions [15]. Schmidt calls this “appropriate obtrusiveness” [41].

Based on these findings about awareness, Schmidt proposes a set of important research questions to be pursued. As an example, he asks the following:

By virtue of which competencies are cooperating actors able to make sense of what others are doing? How does the actor determine what is relevant to his or her own effort?

While the research questions raised by Schmidt are very relevant to both CSCW and Software Engineering. A pair of questions *not* addressed by Schmidt is the following:

How do social actors determine *to whom* should they display their actions? And, *whose* actions should they monitor?

These questions are exactly the questions we address in this paper. They have not received enough analytical attention in general, either by the Software Engineering or CSCW research communities. To some degree, they have been hinted at from a technological point of view, for example, in event notification servers [36], usually through

subscriptions that allow one to define the notifications to receive. Empirical studies of awareness have been limited as well regarding the research questions above, in part because the studies of work practice that helped to establish the concept of awareness focused on the organization of the work in settings with specific features: there was a strict division of labor, all actors were physically collocated, and work tasks required coordination and were highly interdependent. That is, the settings studied (control rooms [3], newsrooms [30], surgery rooms [31], software development war rooms [51] and trading rooms [28]) required individuals to monitor their colleagues’ immediate actions at the same time they were engaged in other activities [31]. These types of places share so many features that they have been called “centers of coordination” [50].

Therefore, we argue that a new analytical focus or “lens” is useful: instead of focusing on the coordinative practices of displaying and monitoring, we should focus on *the work practices by which social actors identify the colleagues who should be monitored and those colleagues to whom their actions should be displayed*. In this paper, instead of taking for granted the social actors involved in the coordination of work through awareness, we unpack *how software developers in their daily work identify this set of actors*. This is necessary to properly understand how collaboration is achieved in software development efforts, and to allow computational support for awareness. The work reported in this paper also provides an understanding of which and how different aspects (e.g., the organizational setting) facilitate the identification of these actors. By answering these questions, we can design better collaboration tools that facilitate the coordination of work, especially, software development work. As we mentioned before, this has not been studied in previous studies of collaborative work, neither in software development nor in other domains.

### 3 RESEARCH SITES AND METHODS

We conducted three qualitative studies at two different large software development organizations. The first field study was conducted during summer 2002, the second one was performed during summer 2003, and finally, the third one was performed during summer 2004. The role of the software architecture in the software developers’ work practices was evident during the three different data collections; therefore, we explicitly tried to collect information about this aspect.

Details about each team as well as the methods used to collect data with each team are described next. Then, we describe how our data analysis was performed.

#### 3.1 Alpha

In this study, the first team has developed a software application called Alpha (not the real name), a software composed of ten different tools in approximately one million lines of C and C++ code. Each one of these tools uses a specific set of “processes.” A process for the Alpha team is a program that runs with the appropriate run-time op-

tions and it is not formally related to the concept of processes in operating systems and/or distributed systems. Running a tool means running the processes required by this tool with their appropriate run-time options. Processes are used to divide the work: Process leaders and process developers, usually work with only one process. Each developer is assigned to one or more processes and tends to specialize in each of these. This is an important aspect because it allows developers to deeply understand a process's behavior and structure, allowing them to deal with the complexity of the code. Process leaders are responsible for reviewing each change made to their process.

The software development team is divided into two groups: the developers and the verification and validation (V&V) staff. The developers are responsible for writing new code, fixing bugs, and adding new features to the software. This group comprises twenty-five members, three of whom are also researchers who write their own code to explore new ideas. V&V members are responsible for testing and reporting bugs identified in the Alpha software, keeping a running version of the software for demonstration purposes, and maintaining the documentation (mainly user manuals) of the software. This group comprises six members. Developers and V & V team members are located in several offices across two floors in the same building, i.e., they were collocated.

The Alpha group adopts a formal software development process [22] that prescribes the steps to be performed by the developers. For example, all developers, after finishing the implementation of a change, are supposed to integrate their code with the main baseline. In addition, each developer is responsible for testing his or her code to guarantee that when the changes are integrated, bugs will not be added to the software. Another part of the process prescribes that, after checking-in files to the repository, a developer must send an email to the software development mailing list describing the problem report (PR) associated with the changes, the files that were changed, and the branch where the check-in will be performed, among other pieces of information.

The first author spent eight weeks as a member of the Alpha team with the sole purpose of collecting data. He did not write any code, tests, or problem reports, but instead made observations and collected information about several aspects of the team, talking with colleagues to learn more about their work. Additional material was collected by reading manuals of the Alpha tools, manuals of the software development tools, formal documents (such as the description of the software development process and the ISO 9001 procedures), training documentation for new developers, PRs, and so on. All Alpha team members agreed to the data collection. Furthermore, some of the team members agreed to be shadowed for a few days. These team members belonged to different groups and played diverse roles in the Alpha team. They worked with different Alpha processes and tools and had varied experience in software development, which allowed a broad overview of the work being performed at the site. Eight Alpha team members were interviewed

during 45- to 120-minute sessions, according to their availability. To summarize, the data collected consist of a set of notes that resulted from conversations and documents as well as observations based on shadowing developers.

### 3.2 Beta

The second field study was conducted in a software development company named BSC. The project studied, called Beta, is responsible for developing a client-server application. The project staff includes 57 software engineers, user-interface designers, software architects, and managers, divided into five different teams, each one developing a different part of the application. The teams are designated as follows: lead, client, server, infrastructure, and test. The lead team comprises the project lead, development manager, user interface designers, and so on. The client team is developing the client side of the application, whereas the server team is developing the server aspects of the application. The infrastructure team is working in the shared components to be used by both the client and server teams. Finally, the test team is responsible for the quality assurance of the product, testing the software produced by the other teams. Most Beta developers were collocated in large campus in the same city, with a few developers located in a different city 1-hour away from the others. In the remainder of this paper, members of the client (server) team will be called Beta client (server) developers.

The Beta project is part of a larger company strategy focusing on software reuse. This strategy aims to create software components (each one developed by a different project/team) that can be used by other projects (teams) in the organization. Indeed, the Beta project uses several components provided by other projects, which means that members of the Beta teams need to interact with other software developers in other parts of the organization.

To facilitate the reuse program, BSC enforces the usage of a reference architecture during the development of software applications. The BSC reference architecture prescribes the adoption of some particular design patterns [23], but at the same time gives software architects across the organization flexibility in their designs. This architecture is based on tiers (or layers) so that components in one tier can request services only to the components in the tier immediately below them [6]. Data exchange between tiers is possible through well-defined objects called "value objects." Meanwhile, service requests between tiers are possible through Application Programming Interfaces (APIs) that hide the details of how those services are performed (e.g., either remotely or locally, with cached data or not, etc.). In this organization, APIs are designed by software architects in a technical process that involves the definition of classes, method signatures, and other programming language concepts, and the associated documentation. APIs are both a technical construct and an organizational mechanism that allows teams to work independently [16].

Regarding data collection in this field study, we also

adopted non-participant observation [34] and semi-structured interviews [38] which involved the first author spending 11 weeks at the field site. Among other documents, meeting invitations, presentations, product requests for software changes, emails, and instant messages exchanged among the software engineers were collected. All this information was used in addition to field notes generated by the observations and interviews. We conducted a total of 15 semi-structured interviews with members of all five sub-teams. Interviews lasted between 35 and 90 minutes. To some extent, an interview guide was reused from the Alpha field study to guarantee that similar issues were addressed. These data were analyzed to understand the role of APIs in the coordination of Beta developers, as reported elsewhere [16].

### 3.3 Gamma

The third team we studied, Gamma, was responsible for developing a mobile application. In fact, this application was a mobile version of the application developed by the Beta team. Because of that, Gamma developers wanted to use, as much as possible, the Beta source-code, but this was not always possible because of hardware constraints in the mobile device they were targeting. Gamma and Beta were part of the same organization.

The Gamma project staff was divided into three major groups: user interface (UI) designers, software developers, and the quality assurance (QA) team. The staff was distributed over five different sites in three countries: North Carolina, US; Massachusetts, US; Beijing, China; Shanghai, China; and Taipei, Taiwan. To be more specific, user interface design and evaluation was performed by six professionals in North Carolina. The implementation was performed in all other sites, distributed as follows: nine developers in Massachusetts, five in Shanghai, five in Beijing, and four in Taipei. The quality assurance team was divided between the US and Chinese sites: three engineers were located in Massachusetts and six engineers in Beijing. The main coordination of the project and the project manager for this project were located in Massachusetts, where all the data were collected.

In the Gamma team, data were collected through document analysis and semi-structured interviews [38]. Among other documentation, emails and instant messages exchanged among the software engineers were collected. We again were granted access to shared discussion databases used by the software engineers. This information was used in addition to notes generated by the interviews. We conducted 17 semi-structured interviews with members of all sub-teams from the different sites: some interviews were conducted face to face, and others were conducted by telephone, with one interview conducted by using instant messaging. We reused some of the questions used in the interviews with Beta team members, but we also explored communication, collaboration, and coordination efforts between their collocated and distributed colleagues, and between the Beta and Gamma team members. Interviews lasted between 20 and 70 minutes. We also interviewed three members of another team whose component provided services to the Gamma appli-

cation, but not to the Beta application.

At the time data was collected in Gamma, developers were finishing the implementation of their prototype, more specifically they were about two weeks away from “feature freeze”. Our interviews, however, focused on the entire software development process, i.e., our interview guide included questions about the requirements of the Gamma software, its UI design and, how testing was already being, and planned to be performed. In our analysis, we focus on aspects regarding the entire software development process instead of focusing solely on the implementation phase, as done in our analysis of the Beta team.

### 3.4 Data Analysis

All of the data that was not already text was transformed into text for analysis. E.g., interviews and handwritten field notes were transcribed into text. The complete data was input into a software tool for qualitative data analysis. After that, we used coding techniques to make sense of the data we collected: interviews and field notes were coded to identify categories that were later interconnected with other categories. Using this approach, we identified the concept of the awareness network – the network of actors whose actions need to be monitored by an actor and those to whom this actor needs to make his or her own actions visible – and reported our first results in [14]. Later, the third dataset, Gamma, was integrated into the same set of categories and relationships already identified in order to further investigate the awareness network concept. This integrated analysis of the three different datasets is reported in this paper.

At this point, it is important to distinguish the focus of analysis in each dataset. In the Alpha team, developers were performing maintenance of the Alpha application, i.e., at the time of the study there already were previous versions of the Alpha software, i.e., Alpha developers were making changes in the next version of their software. Furthermore, each change in the software was associated with a change request in the bug-tracking tool (see section 4.1). This means that, in this project, we were concerned with the work practices adopted by software developers that lead to the identification of their awareness networks during the work required to implement each change<sup>2</sup>. Meanwhile, the Beta team was developing the first version of their client-server application and as a consequence chose not to enforce strict bug-tracking processes. Therefore, our analysis focused on aspects related to the implementation of the Beta software. Also, our analysis of the awareness network in the Beta team was broader than in the Alpha team because it encompassed the entire implementation phase. Again, this means that our data collection and observation included data about it, i.e., our questions were asked regarding the beginning of the implementation phase. Finally, in the Gamma team, we focused our analysis on the entire software development process. In other words, while the data was collected during the implementation phase, we report here aspects of the awareness network identification

<sup>2</sup> This is similar to what was done by Cataldo and colleagues [9].

that are related to past software development phases (requirements and design) and future ones (testing).

The following sections describe the work practices of the Alpha, Beta, and Gamma teams, how their developers identify their awareness networks, and the organizational factors that influence these practices.

## 4 THE AWARENESS NETWORK IN THE ALPHA TEAM

### 4.1 Task Assignment

For accountability purposes, all changes in the Alpha software need to be associated with a problem report (PR). Among other pieces of information, a PR describes the changes in the code, the reason for the changes (bug fixing, enhancement, etc.), and who made the changes. An Alpha developer is delegated new tasks by being assigned to work with one or more PRs. These PRs are reported by other team members. Whoever is filling in the PR is responsible for filling in the field “how to repeat,” which describes the circumstances (data, tools, and their parameters) under which the problem appeared. When software developers report a PR, they also might divide a PR into multiple PRs that achieve the same goal. This division aims to facilitate the organization of the changes in the source code, separating PRs that affect the released Alpha tools from those PRs that affect tools or processes not yet released.

As mentioned in the previous section, each developer is assigned to one or more processes and tends to specialize in that process. A manager will follow this practice and assign developers to work on PRs that affect “their” respective processes. However, it is not unusual to find developers working in different processes.<sup>3</sup> In this case, Alpha developers need to identify and contact the process owner to find out whether there is a problem in the process.<sup>4</sup> If there is a problem, developers will start working to find a solution to this problem. Even if the problem is straightforward, before committing their code, Alpha developers need to contact process owners to verify, through a code review (a prescription of the Alpha software development process), whether their changes in the process are going to impact the work of these process owners. Code reviews are performed by process leaders whose processes are affected by the changes in the PR. If the changes involve more than one process, a request for a code review has to be made to the owner of each process affected by those changes.

<sup>3</sup> This might happen due to various circumstances. For example, before launching a new release, the entire workforce is needed to fix bugs in the code; therefore, developers might be assigned to fix these bugs no matter where they are located. Or, a developer who already started working on a bug, because it seemed to be located in his or her process, might later find out that the bug is located in a different process. In this case, it is easier to let that developer continue to fix the bug due to the time already spent in understanding it, than to assign it to a different developer at that point.

<sup>4</sup> Sometimes bugs are reported because of an abnormal behavior that might be considered a problem; the role of the developers in this case is precisely to find out whether there is a problem. This happens due to the complexity of the Alpha code and the lack of domain knowledge of Alpha software developers [12]. In this case, developers discuss the issue face-to-face and/or by email, and a PR is not inserted in the bug-tracking tool until the existence of a bug is confirmed.

### 4.2 Identifying Who to Contact

The need to contact process owners means that the developer working with the PR needs to identify the owner of the process being affected. This is not a problem for most developers, who have been working in the project for a couple of years and already know which developers work on which parts of the source code. In contrast, developers who recently joined the project face a different situation because they lack this knowledge. To handle this situation, newcomers use information available in the team’s mailing list. The software development process prescribes that software developers should send email to this list before integrating their changes in the shared repository. Developers thus associate the author of the emails describing the changes with the “process” where the changes were occurring: Alpha team members assume that if one developer repeatedly performed check-ins in a specific process, it was very likely that he or she was an expert on that process. Therefore, a developer needing help with that process would know who to contact for help. According to Alpha-Developer-04:

If you are used to looking at the headlines and know that [tool1] stuff seems to always have [Alpha developer1]’s name on it and all of a sudden you get a bug, for us with the GUI because you can get it from any point, I could end up with a GUI bug that ends up being [tool1]-ish in the PGUI and what do I do? I don’t understand why this thing behaves the way it does but most of those PRs seem to have [Alpha developer1]’s name on them. So you go down and see [Alpha developer1]. So by just reading the headline and who does what, you kind of get a feeling of who does what, which isn’t always bad. (...) [Alpha developers2] does [tools2] sort of stuff and although I have never had to talk to him about it, but if I run into a problem, by reading the email or seeing them, he tends to deal with that kind of stuff so they [the broadcast email messages] tend to be helpful in that aspect as well. If you have been around 10 years, you don’t care, you already know this. I have only been here two years and that stuff can make a difference—who you ask the question to when you get in trouble.

This quote illustrates how new members have difficulty in identifying who to contact for help; their awareness network is unknown. This also shows how software developers use an organizational guideline (broadcast emails for each check-in) to glean the necessary information.

### 4.3 The PR Work

After having his or her changes approved by the process owner(s), a developer fills in the other fields of the PR, describing not only the changes made in the code (through the *designNar* field, for example), but also the impact these changes are going to have on the V&V staff.<sup>5</sup> The information about the impact on the V&V staff is re-

<sup>5</sup> Process leads also use information about changes in the code to assess the impact of the changes in the software architecture during code reviews.

corded in two PR fields: (i) the “how-to-test-it” field is used by the test manager, who creates test matrices that will later be used by the testers during the regression testing; and (ii) another field that describes whether the Alpha manuals need updating. The documentation expert uses this information to find out whether the manuals need to be updated, based on the changes introduced by the PR. In some cases, developers are even more specific:

Developers will be very helpful and they will say “Figure 7-23 in the [tool] manual needs to be changed.” If they do that, it makes my job easier and I appreciate it, but I don’t expect it.

In short, problem reports facilitate the coordination of the work among Alpha team members. They provide information that helps team members understand how their work is going to be impacted, which is useful for different members of the team according to the roles they are playing.

#### 4.4 Writing Emails

To conclude the work required to make changes in the Alpha software, developers need to inform their colleagues that they are about to commit their changes to the shared repository. This is done by sending an email to the rest of the team. These emails are necessary due to the lack of modularity of the Alpha software: a change in one particular “process” could impact all other “processes.” According to a senior Alpha developer:

There are a lot of unstated design rules about what goes where and how you implement a new functionality, and whether it should be in the adaptation data or in the software, or should it be in [process1] or should it be in [process2]. Sometimes you can almost put functions anywhere. *Every process knows about everything*, so just by makefiles and stuff you can start to move files where they shouldn’t be, and over time it would just become completely unmain-  
tainable. ... *yeah, every process talks to every other one.*  
[emphasis added]

We discuss later in section 7.4 how the structure of the Alpha software, in particular, its non-modular software architecture, influences the strategies used by software developers to identify their awareness networks.

#### 4.5 Reading Emails

Emails exchanged among team members are also used by software developers to find out whether they have been engaged in parallel development. Parallel development happens when several developers have the same file checked out and are simultaneously making changes in this file [42]. Note that, in the Alpha team, if a developer, John, is engaged in parallel development with another developer, Mary, and Mary already checked in her changes in the main branch before John did, John will necessarily have received an email from Mary about her check-ins. By reading these emails, John will be aware that he is engaged in parallel development with Mary

because her email describes, among other things, the files that have been checked in<sup>6</sup>. In this case, John is required to perform an operation known in the Alpha team as a “back merge.” This operation is supported by the configuration management tool adopted by the team and is required before a developer can merge his or her code into the main branch.

Parallel development happens because the Alpha software is organized in such a way that parts of it contain important definitions that are used throughout the rest of the software. This means that several developers constantly change these parts in parallel; back merges thus are performed fairly often:

It depends on ... there are certain files, like if I am in [process1] and just in the [process2] that [back merges] is probably not going to happen, if I am in the [process3] there is like ... there is socket related files and stuff like that. I think [filename] and things of that sort. There’s a lot of people in there. The probability of doing “back merging” there is a lot higher.

To avoid back merges without avoiding parallel development, Alpha developers perform “partial check-ins.” In a partial check-in, a developer checks in some of the files back to the main repository, even when he or she has not yet finished all the changes required for the PR. The checked-in files are usually those that are changed in parallel by several developers. This strategy reduces the number of back merges needed and minimizes the likelihood of conflicting changes during parallel development. In other words, Alpha developers employ partial check-ins to avoid being affected by other developer’s changes in the same files because these changes can generate additional work for the developers.

## 5 THE AWARENESS NETWORK IN THE BETA TEAM

### 5.1 The Organizational Context

As mentioned previously, applications developed in the BSC organization should be designed according to a reference architecture based on layers and APIs, so that components in one layer could request services only for components in the layers immediately below them through the services specified in the APIs. By using this approach, changes in one component could be performed more easily because the impact of these changes is restricted to a predefined set of software components. In addition, changes in the internal details of the component can be performed without affecting this component’s clients. As a consequence of this approach, it is not necessary to broadcast changes to several different software developers, but instead just to a small set of them. That is, by decoupling software components, it is possible to facilitate the coordination of the developers working with these components [11, 41].

<sup>6</sup> Note that by reading emails, Alpha developers can both identify their awareness network and, at the same time, to become aware of their colleagues’ actions. In our case, we are interested in the identification of the networks, but this is only an analytical distinction.

Unfortunately, organizational factors decrease the effectiveness of this approach. For example, the large-scale reuse program adopted by BSC leads Beta developers to interact with developers in different teams who can be located anywhere: in the same building, in different cities, or even in different countries. This is necessary to allow software components to be reused within the organization and, therefore, to reduce software development costs. However, due to the size and geographical distribution of the organization, this approach was problematic. During our interviews, we found out that Beta server developers do not know who is consuming the services provided by their components, and Beta client developers do not know who is implementing the component on which they depend. Because of that, developers do not receive important information that affects their work (e.g., important meetings they need to attend).

This problem is aggravated by the young age of the project, according to Beta Developer-15:

When you sit on a team for two years, you know who everybody is. Even peripherally you know who people are. So if we had to get answers about [*another BSC product in the market for years*], we have so many people on the team who were on the team for so long [*a*] period of time they can get the answer immediately. They know who the person is even if they have never met them. We don't have that in this group because it takes time for those relationships to develop ... like I talked to so and so and talked to so and so and so on. You only have to go through that once or twice because once you have gone through that you know the person. I think part of that frustration is how you spin up those relationships more quickly. I don't know if you realize this but this team has only been in existence since last year. So it is a ten-month-old team.

In short, Beta developers have difficulty identifying who they need to contact to get their work done, and they acknowledge that this is problematic. A developer, for instance, reported talking to up to 15 people before finding the right person:

Interviewer: "So have you experienced this problem?"

Beta Developer-15: "Totally. That is what I have said. I am kind of merciless in trying to find the right person. I have shotgunned up to four or five people at once to say 'do you know who is responsible for this?' and then gotten some leads and followed up on those leads and talked to as many as 10 to 15 different people."

Another developer complained about the need to simplify the "communication channels" in the organization to avoid having to interact with different managers to find out who was the person responsible for implementing a particular software component. This same developer reported that one of the teams providing a component to his team is not even aware of his team's need. On another occasion, a developer tried to find out whether

she could use a particular user-interface (UI) component. The UI designer working with her indicated a developer in Japan who was using this same component. It was this Japanese developer who recommended to her another software developer, back in the U.S., who was implementing the UI component she wanted! Finally, a developer suggested that a database containing information about who was doing what in the organization was necessary: "sometimes you wanna talk to a developer ... the developer in the team who is working in this feature [that you need]."

Architects and managers also recognized this situation as problematic:

The problem with that [not knowing who to contact] too is that there is another case where people are thinking that there is someone else doing something [but] when push comes to shove and it gets pushed on to you, it is an empty void because they don't stand up and say that they have tried to identify their server counterpart and my client counterpart and there is not one. We have a problem here.

## 5.2 Identifying Who to Contact

In order to identify who they need to contact, developers adopt different approaches. First, they rely on their personal social networks and "activate" them as they see fit, for instance, by sending email to colleagues within and outside the team asking for help. Managers also play an important role in this process due to their larger social networks. Beta developers contact them so that these managers can identify the person they want to find.

Another approach to identify the relevant person to be contacted is to leverage their knowledge about the software architecture. In one occasion, a client developer "followed" his technical dependency in order to switch teams: his software component had a dependency on a component provided by the server team, who actually had a dependency in a component from the infrastructure team, who depended on an external team's component. To simplify the communication channels and make sure that the client team would have the component it needed, the manager of the client team decided to "lend" one of the client developers to the external team.<sup>7</sup> By doing so, the manager, to some extent, could guarantee that the needed services would be implemented and that the person doing this work would be someone familiar. In other words, by doing that the manager could guarantee that his awareness network was stable.

Identifying whom to contact is a problem in the entire BSC organization. Indeed, BSC managers create a discussion database that developers can use to identify the people necessary to answer their questions. However, due to the large number of databases already in use, managers have to slowly convince BSC developers of the importance of this particular database:

The management team is really trying to socialize the idea that [the discussion database] is the place to go when you

<sup>7</sup> In software engineering, artifact dependencies (such as the ones that exist among the components of a software system), often imply dependencies among software developers [24, 48].

have a question and you don't know who can answer it. They are really trying to socialize that people should give a scan to it every once in a while to see if they can help and answer a question. The amount of traffic there has picked up quite a bit in the last couple of months, especially in the past couple of weeks. My team has not gotten that message a hundred percent yet. There is a tool for it and a place to go that I have had a lot of success with when I use it; it is just that the message has not gotten out yet that that is the place to go. One of the things that happens when you have so many databases [is] it takes a while for one to emerge as the place to be. This is turning out to be the place to be.

Not everything is hectic in the BSC organization, though. An organizational aspect facilitates the identification of the awareness network: the API review meetings. Within the Beta team, these meetings are scheduled to discuss the APIs being developed by the server team. The following people are invited: API consumers, API producers, and the test team that eventually will test the software component's functionality through this API. In addition to guaranteeing that the API meets the requirements of the client team and that this team understands how to use it, this meeting also allows software developers to meet. After that, the server team provides APIs to the client team with "dummy implementations" to temporarily reduce communication needs between them, thus allowing independent work. This approach is useful only in some cases, due to the time that passes between API meetings and the actual implementation of the API. In the meantime, changes in developers' assignments may cause communication problems because developers do not know about each other anymore. In short, changes in assignments change the awareness network, thereby making the work of software developers more difficult to coordinate.

### 5.3 On the Effectiveness of Notifications

Beta developers have an expectation that major changes in the software are preceded by notifications, so that everyone is informed about changes that could affect their work. In fact, developers reported warning their colleagues of major changes in the code and their associated implications. This is done in group meetings, which provide an opportunity to developers to inform their teammates. Developers also inform their colleagues on other teams. For instance, server developers inform the installation team of new files being added or removed so that the installation procedures can be updated with this information. In other cases, Beta server developers may negotiate changes with client developers in APIs that existed between the teams before actually performing the changes.

However, the usefulness of these notifications is contingent upon knowing who to contact. As discussed in the previous section, not all Beta developers know their awareness networks, and therefore are not able to provide and receive important notifications. For instance, Beta-Developer-15, when asked about a specific situation

regarding notifications he was supposed to receive, but did not receive, reported the following:

Let me give you an example. Our database developer [name] had certain files that were used to create databases. He changed the names of the files at one point so we lost some time while people were trying to deploy because they went to follow the instructions that I had written and they could not find the files that I was telling them to run. ... But that is the flavor of the type of thing I am talking about.

Because developers can miss important information, a strategy adopted by this same developer is to read everything to find out what could impact him:

Interviewer: "In your particular case, have you not received an email that you should have received? And because you did not receive it, have you wasted one day of work, for instance?"

Beta Developer-15: "Partly that. *I sort of make up for that by reading everything.* Obviously, it is not a generically good solution because it means that you waste a lot of time. I basically stay in a hyper alert state constantly looking for things that impact me. The problem is that you read through a lot of things that you are not really interested in. I have reviewed a lot of these design documents [that I mentioned earlier] and I probably don't ever have to necessarily read but I did not know if there was anything in there that was relevant to installation. ... Part of it is attention, being able to remind somebody that you are interested in what it is that they are doing." [emphasis added]

This quote clearly illustrates one of the problems of the misidentification of awareness network: extra work. In this case, Beta-Developer-15 has to read all email notifications because he does not know who can affect him.

Other developers are similarly concerned about receiving too many notifications about things that are not relevant to them, especially when dealing with discussion databases. That is, they are concerned about not being in one's awareness network and still receiving notifications of changes. According to Beta Developer-13:

I think that in the beginning when it [the discussion database] was small, we used to go in everyday, at least I did, and look for new documents and keep updating. Now it is like, if someone has sent me an email that said that they have a related document and here is a link, that is when I go to it. Because otherwise it is massive amounts of things and I cannot even make sense of it and how it is relevant to me.<sup>8</sup>

Even if the notifications are delivered to the right personnel, notifications are useful only to some extent: once an API is made public, control of who is using it is lost, and therefore notifications are no longer necessary be-

<sup>8</sup> This problem occurred because the BSC was already using several different databases, which were not organized nor updated often. This is a common problem reported by Beta developers.

cause these APIs cannot change. As described by a server developer (Beta Developer-10):

We have latitude to change it [the API] as long as we are talking about an unpublished or semi-private API. If it is a contract between us and the client people, we probably have more latitude to change it and therefore they can trust it a little less than if it was a published API. At that point it would be very difficult to change it because people would be relying on ... right now we control everything that has a dependency, we control all the dependencies because the only people who are using the API are our own client teams and test teams and we can negotiate changes much easier than if they were external customers that were unknown to us or people in the outside world who we don't control and who also could not readily change their code to accommodate our API changes. We would have to go about carefully deprecating, evolving ... some features.

## 6 THE AWARENESS NETWORK OF THE GAMMA TEAM

### 6.1 The Organizational Context

As mentioned before, the Gamma and the Beta teams were part of the BSC organization. However, in contrast to the Beta team, the Gamma team did not have a reference architecture to comply with because its application was targeted to a mobile device with severe resource constraints. BSC's reference architecture aimed at client-server applications. Accordingly, APIs in the Gamma team were not as relevant as in the Beta team from an architectural point of view. Furthermore, APIs in the Gamma team were not team boundaries, as they were in the Beta team, that is, the API boundaries were not aligned with organizational boundaries [16]. Therefore, API design review meetings and other API-related problems were not as relevant. A Gamma developer located in Taipei, who was consuming services from an API, reported that it was easy to coordinate his work with the person who was implementing the API services:

Informant: "... and then I'll send her an email that, 'Is it okay for me to – is that ready for me now?' Then she will tell me that, 'Okay, it's ready,' or 'Sorry, it's not implemented yet.'"

Note that this developer is talking about another developer located in Massachusetts, suggesting that geographical distribution was not a cause of concern for these developers [16]. In another occasion, a different Gamma developer informed us about the effectiveness of notifications about changes in the APIs:

Researcher: "So basically, in case this – assuming that she has to make a change in API, so is she – does she send you an email or something telling that she's changing those or...?"

Informant: "Oh, no. She'll tell me. Well, because the problem is if she actually changed the API, she's going to break my code; my code won't compile, so..."

Researcher: "She's going to send you an email or ping you or even stop by or something?"

Informant: "Yeah, exactly. And I'd have to search through my code and – to make those changes and know that my code wasn't going to compile until it (inaudible) with hers."

In contrast to the Beta team, notifications were effective in the Gamma team because Gamma developers knew who to contact, that is, they were mostly aware of their awareness networks.

Some of the observations in the Gamma team, however, were similar to the Beta team. For instance, developers had an expectation that major changes in one's code that break their colleagues' code would be publicly announced.

In the following section, we focus on aspects of the Gamma team that are relevant for our analysis: the identification of the awareness network during the entire software development process.

### 6.2 The Identification of the Awareness Network

In the Gamma team, our analysis focused on the broader software development process instead of focusing on a specific task (as in the Alpha team) or on solely in the implementation phase (as in the Beta team).

As in most modern software development projects, members of the Gamma team needed to discuss the user-interface of its software. This was an internal discussion because there was a group of user interface designers within the team located in Raleigh, N.C. This discussion was an iterative process in which both developers and UI personnel made suggestions about how to implement the design. According to Gamma-developer-02:

Before the design changes—the design was there and we were implementing it and going, "Oh, this isn't going to work and we need to change this and what-not," and we had a lot of design meetings and changed the specs and then they were playing catch-up for awhile trying to get those changes into the specs and now I think the specs are done and we have to get back in and do the changes.

Note that by the time our data collection took place, according to this same developer, most of the design work and discussions were finished. At that time, software developers were implementing the UI design itself. In addition, the discussion among these different team members occurred using a variety of media, as the team members saw fit. For instance, major design changes proposed by the implementation team required meetings involving both developers and UI designers who needed to agree upon a solution, whereas minor changes were simply negotiated between the developer and UI designer responsible for the feature being changed. In any case, design changes were reflected back into the UI specifica-

tion for future reference.

We usually contact them [the UI team] through email; occasionally through [the instant messenger tool] and for the most part we hash out, either in a meeting, in a teleconference or in an email thread—we'll hash out a specific design point and then, presumably, it will get put into the official spec[ification] in the official database and then we can go back and reference it.

Other teams were involved in the design phases as well. For instance, Gamma-developer-05 was involved with the implementation of the security aspects of the application. According to him:

I also have to interact with a database [name of technology] team in California because the encryption they gave us wasn't strong enough and we requested for the next release a stronger encryption.

We also observed that the members of the Gamma team dealing with the implementation were also concerned with managers and members of the quality assurance (QA) team.

Researcher: "So do you guys have a teamroom or something like that?"

Gamma-developer-04: "Yes. It's a [shared] database. ... one thing that we do use a teamroom for is if people have deliveries to make, in terms of feature work, they'll put a note in the teamroom. And it's a response to a specific document. So I think that's more for managers. At this point we're two weeks away from feature freeze, so they want to know where things are and what's going to be in each build so that—number one, the managers just know that we're making progress toward feature freeze and they'll know—they'll be able to see if things are in trouble or not. And the second reason being that the QA team needs to know specifically what features they can test so they're not writing [bug reports] that are just irrelevant because features haven't been implemented yet."

In other words, developers specifically use the teamroom to provide information to managers and QA team members, that is, they use a tool to keep their colleagues informed about their work. It should be noted that this teamroom has information about who writes information into it, but no information about who reads the information from it. Therefore, developers provide information for the QA and managers teams, without knowing which specific individuals were using this information. In fact, we have additional evidence that most Gamma team members who were implementing the application were not aware of the names of specific individuals of the test team.

Based on our interviews, we noticed that interactions between Gamma developers and members of the UI team were mostly finished; interactions between the Gamma

and security teams were still going on, since Gamma was waiting for features from the security team; and, finally, interactions between Gamma and the QA teams were mostly nonexistent. Members of these teams were coordinating their work through a teamroom. In other words, the awareness network of Gamma team members changed according to the software development phase. In the past, it involved UI designers more actively; currently, this is true to a lesser extent. Security and other infrastructure teams and managers were also part of the awareness network during the data collection. And, finally, the QA team was not a relevant part of the awareness network at the moment, but they were going to be once the implementation team finished their work. Here again, the software architecture plays a major role: not all software developers were involved in implementing UI or infra-structure concerns—only developers working on particular parts of the software architecture. That is to say that the software architecture, to some extent, influenced the awareness network of Gamma developers. The QA team, in contrast, was not influenced by the software architecture because all parts needed to be validated.

## 7 DISCUSSION

### 7.1 On The Conceptualization of Awareness

It is important to point out that the concept of awareness has caused some controversy within the CSCW community, with some researchers arguing that it has been more harmful than useful [31]. As noted in the introduction, a large body of previous work has adopted the description proposed by Dourish and Bellotti [18]: "an understanding of the activities of others, which provides a context for your own activity." This description has been interpreted in such way that awareness is then reduced to simply information that is shared among social actors. Based on such a conceptualization, some researchers have created classifications of types of information that are useful in supporting awareness [26].

Such codifications benefit designers interested in providing tool support for collaborative activities. For instance, event notification servers [35, 36] allow one actor to subscribe to the information that he or she is interested in receiving, while other information sources provide information that it is sent to this server. In other words, this approach is based on two assumptions. The first assumption is that social actors actively subscribe to information that is relevant for them. The second is that one knows which information sources should provide information to those interested. As our ethnographic data suggests, neither assumption is always true. Hence, we were motivated to re-examine the work practices that support awareness in distributed settings.

### 7.2 Awareness and Socio-Technical Congruence

Recently, the socio-technical congruence approach [7, 9] has gained attention in the literature of collaborative software development. Congruence is based on a match between "the coordination requirements established by the dependencies among the tasks and the actual coordination activities carried out by individuals". Often, these

coordination activities are associated with communication. We believe the concept of awareness and findings from previous CSCW research can shed light into interesting aspects of the concept of congruence.

As we discussed in the introduction, awareness has been conceptualized as a set of work practices used by competent social actors to successfully coordinate their work. These practices involve both monitoring and displaying of actions, and actors appropriately choose the degree of obtrusiveness to be used to monitor and display their actions according to the context at hand. Seminal studies of work practices [27-29] have pointed out different mechanisms used by actors, including pointing out an object, gazing, moving towards an object, stressing words in a conversation with a third person, and so on. In these cases, coordination is achieved without direct communication between the actors involved and often, no trace of such coordination is left. By understanding the concept of awareness, it becomes clearer that coordination does not imply direct communication among the parties. This result is aligned with results in organizational theory [52, 56]. A further implication is that care must be taken if the concept of congruence is operationalized: *to assume that coordination is based solely on direct communication between the parties is a mistake*. For instance, in the Alpha team, developers provide information into the PR about “how to test” the change they just wrote. When this information is filled the software developer does not even know who is the QA team member who is going to use it. Even in distributed teams similar situations occur: in the Gamma team, developers write information into the shared database about the code they are finishing and that information is used by members of the QA team without developers knowing which member of the QA team was going to use it. In short, congruence gaps [55] or socio-technical clashes [2] need to be carefully investigated: the simple fact that communication between developers, in collocated or distributed contexts, is not taking place, does not imply that these developers are not *successfully* coordinating their work.

### 7.3 The Fluidity of the Awareness Networks

As mentioned before, the concept of awareness is associated with work practices used by competent social actors to understand the state of their colleagues’ work and to successfully coordinate their work. However, in order to do so, actors need, before anything else, to identify which other actors are relevant to be monitored and to have their actions displayed to. Based on our ethnographic data, we illustrate different approaches used by software developers to find out about that, including writing and reading emails (Alpha team), “following” dependencies (Beta team), reading (Beta team) and writing (Gamma team) information into databases, among others.

It is worth noting that awareness networks are fluid and context-specific. That is, these networks easily change components (the software developers involved) and size. For instance, once an Alpha developer starts working in a PR, his or her awareness network is limited to the owners of the processes that the PR involves. This is necessary because these owners can provide information about the

potential problem investigated in the PR (they are the ones who can answer the question: “is it really a problem?”). Note that developers do not know beforehand all the processes involved in the change—a common situation in software systems [47]. Therefore, this network might change as a software developer explores the problem described in the PR. When developers need to fill in the PR fields to complete their work, their awareness network includes the V&V team members who will be affected by their changes. In this case, the identification of the awareness network is facilitated by the PRs because these artifacts already provide useful information about the impact of changes. Finally, before checking-in their changes, software developers’ awareness networks become the entire software development team as they need to broadcast their changes to their colleagues. This is necessary because Alpha’s software architecture is non-modular and a change in a process can impact all other processes. During this whole process, if developers are engaged in parallel development, their awareness network includes the other developers who are changing the same files. To deal with this situation, developers perform partial check-in’s of files that are more likely to lead to parallel development. In this case, software developers use their knowledge about the software architecture (the files that exhibit high degree of parallel changes [42]) to reduce the size of the awareness network and, by doing so, reduce their coordination efforts.

The same fluidity can be observed in the Beta team, but now on a different scale, i.e., instead of being context-specific (or PR-specific), the awareness network changes according to the overall implementation, which can arguably be associated with either API-implementation or API-consumption. In this case, changes in developers’ assignments lead to changes in their awareness networks. In addition, when new developers start to reuse a software component through its API, this means that the awareness network of both the component’s provider and consumer increases. To be more specific, APIs go through a publication process: they are initially private (without clients), then they are made semi-public (they have internal clients), and finally they are publicized (external clients can use it). Private APIs can be changed without a problem because no one is affected. Semi-public APIs require changes to be negotiated to minimize their impact on their clients. Finally, public APIs cannot be easily changed; they have to go through a slow process of change in which API services are somehow marked to indicate that API consumers should stop using them. As an API goes through this publication process, the awareness networks of the developers implementing the API expand: initially the awareness network is small because almost no one is affected, but in the end it becomes so large that it is impossible for an actor to account for all its members. As the awareness network expands, software developers’ work practices need to change as well to accommodate this situation.

Finally, in the Gamma team, it is also possible to observe changes in the awareness network, but these changes are related to the specific software development

phase. During the requirements, the UI team was a major part of software developers' awareness network because changes in the design specification lead them to change their work, and, also suggestions of changes made by developers would impact UI designers. During the overall project phase, infra-structure teams (like the database) provided services to the Gamma team, therefore they were part of the awareness network of these developers. Finally, the QA team was part of the awareness network of software developers towards the end of the implementation phase, when these developers started finishing the implementation of certain features, therefore they needed to inform QA team members which features needed testing. Note also that Gamma developers provided information to the QA team without knowing which individual team members would use this information. In other words, the awareness network of developers included the QA team as a whole, instead of specific members of this team.

Note, however, that the fluidity of the awareness networks in the Alpha, Beta and Gamma teams is different because of the scope of the analysis we performed. More specifically, whereas the awareness network changes somewhat rapidly during the course of work on a PR for Alpha developers, it changes slower in the Beta team because it is related to the overall implementation phase, and, finally it changes even slower in the Gamma team because the awareness network is largely associated with the software development phase in due course. Accordingly, changes in the Alpha developers' network are temporary (they last only while the PR work lasts), whereas changes in the Beta and Gamma teams are more permanent, at least until the next change in assignments in the Beta or the next process phase in Gamma.

## 7.4 Factors Influencing the Awareness Network

The data clearly present how three different factors influence the awareness network: the organization-wide reuse program, the young age of the project, and, finally, the software architecture.

The organizational reuse program in the BSC corporation influences the size of the awareness network: a Beta developer can need information from any other software developer in the organization, if the first developer's code depends on the second's. The result of this approach is that a software developer's awareness network could potentially be any software developer in the organization. API team meetings alleviate this situation because they allow component providers and consumers to meet; however, changes in team membership make the situation vulnerable again. To deal with this problem, Beta developers adopt approaches to identify their networks (social networks, databases, etc.) and broadcast messages, but this causes complaints about information overflow that is not related to one's work. In other words, Beta developers may receive notifications from developers who do not belong to their awareness network. This situation is identified throughout the entire organization. Overall, this situation was not problematic in the Gamma team because the hardware constraints they had to adhere to led

to fewer interactions between the Gamma team and other teams in the BSC organization.

The second aspect that influences the identification of the awareness network is the software developers' experience in the project. Whereas the Alpha project had been going on for more than nine years at the time of the study, the Beta project and the BSC organizational reuse program existed for little more than nine months! As one developer pointed out, this was not enough time to allow software developers to establish the social connections among themselves required for the accomplishment of their work. Similarly, novice Alpha developers mentioned the importance of knowing who to contact in the project in order to finish their work without impacting their colleagues.

Finally, the software architecture is the third factor that influences the awareness network. Alpha software developers recognize that the Alpha software architecture is not modular, and as a result, a change in one software process can affect several other processes (and their developers). In contrast, Beta software has an architecture defined according to best practices in Software Engineering with controlled dependencies through layers, APIs, and so on. This architecture, called modular, implies a small number of developers being impacted by changes. Gamma's architecture was also modular, although not as modular as Beta due to constraints in the mobile device. On the one hand, a non-modular architecture leads to larger awareness networks, and as a result, specific coordinative practices: the displaying of actions is done by email broadcasts and PR fields, whereas the monitoring is performed by reading emails. On the other hand, modular architectures lead to more manageable awareness networks, which could not be fulfilled in the Beta team due to organizational factors. In this case, developers also display their knowledge about the software architecture when they "follow the dependency" to find out to which team they should switch in order to provide the necessary services. Note that the influence of the software architecture result is not surprising because this influence has long been recognized to affect the coordination of the work [9, 11, 16, 24, 41, 48]. *What is more relevant here is finding out how software developers make use of that information to facilitate their work.*

## 7.5 Managing Awareness Networks

Another important aspect that deserves attention is software developers' effort to manage their awareness networks. This aspect can be illustrated in all three teams. In the Alpha team, the more experienced software developers relied on their knowledge about the software architecture to avoid engaging in parallel development, because this obliges one to monitor other software developers, i.e., this enlarges a software developer's awareness network. To reduce their networks, Alpha developers perform "partial check-ins". Similarly, the Beta team manager, as discussed, proposed to "lend" one of his developers to another team so that he could work on the feature Beta developers needed. By doing that, the Beta team would avoid changes to that particular piece of his awareness

network: as long as its developer was in the other team, team members would know whom to contact about the particular feature. Finally, the Gamma development sub-team used a database to “communicate” the conclusion of their features for the Gamma QA sub-team. Through the database, developers could provide information to the QA team, without needing to be familiar with the *individual* QA developer testing their code. This is again similar to what was done by Alpha developers who filled PR information that was relevant to QA members, without knowing which QA team member was going to use the information. In all three cases, it is clear that software developers try to manage their awareness network to avoid engaging in additional coordination.

### 7.6 Problems from Failing to Identify Awareness Networks

The reason why software developers in the three teams we studied were actively managing their awareness networks is because not knowing their networks was very costly. Our data clearly stresses the importance of this identification and the associated cost of misidentification. This cost of lacking awareness is backed up by other researchers [20]. Taking an instance from our data, observations from the Beta team suggest that when the awareness network is misidentified, the collaborative effort is severely damaged: most of the problems faced by Beta developers (delays in their work, extra-work, uninteresting notifications, notifications overflow, missing notifications, and so on) are due to difficulty in identifying their awareness network. To deal with this problem, these developers need to adjust their work practices accordingly. As we discussed before, Gamma developers, although located in the same organization, did not see this identification as a problem because code dependencies and organizational boundaries were not aligned [16]. Similarly, in the Alpha team: due to the duration of the project, most developers already knew their colleagues’ expertise and, therefore, their awareness networks. The only exception was newcomers, who did not know this information, but who used email to identify the process leaders.

### 7.7 Comparing Awareness and Expertise Networks

Some of our colleagues have observed a similarity between the concept of an *awareness network* and the concept of an *expertise network*. Therefore, we developed a comparison to clarify the concepts.

Collecting some descriptions of awareness and expertise finding (and associated terms) provides a starting point. There is the Dourish and Bellotti description of *awareness* as “an understanding of the activities of others, which provides a context for your own activity [18].” In the introduction to this present paper, we described an *awareness network* to be “the network of actors whose actions need to be monitored by an actor and those to whom this actor needs to make his or her own actions visible.” Ehrlich explains that “an *expertise locator* provides a valuable tool for individuals to develop awareness of ‘who knows what’ and to reach out to people across the organization [19].” Her description is very con-

sistent with Ackerman and colleagues who explain that “*expertise finders*, or expertise recommenders, are a form of recommendation system ... but expertise finders point people to other people. [1]” Nardi and colleagues describe *intensional networks* as social networks that individuals maintain in order to effectively come together to achieve a task, a very dynamic and fluid team [39].

Reviewing these descriptions begins to reveal some similarities and differences at a high level. Knowing whose actions need to be monitored and to whom one’s actions should be made visible is critical in awareness. Knowing who to contact and speak with is critical in expertise location. In both cases, there is a kind of knowing about others that is essential. In the context of this paper, and the awareness network concept, the knowing is about others’ activities and responsibilities in an on-going task, e.g., a software development or maintenance task. With respect to expertise networks, the knowing is about others’ aggregated or historical achievement in a wide range of contexts and activities. In short, the former evokes a characteristic of specificity while the later evokes a characteristic of generality.

We can examine the concepts along several dimensions emphasizing the extremes, where prototypical differences are clearer. *Timeframe and scope*. Expertise networks are intended to cover organizational timeframes whereas awareness networks exist in task and project timeframes. Expertise networks cover careers and multiple projects while awareness networks cover members in a team and one or a few projects. *Purpose*. Expertise networks focus on general abilities whereas awareness networks focus on specific activities. *Size*. Expertise networks are enterprise wide while awareness networks are team sized.

Similarly, reflecting on objectives, some contrasts emerge at the extremes. Expertise networks reflect the developing and maintaining of an information repository (the expertise network) while awareness networks reflect the carrying out of a specific task. Expertise networks require eliciting tacit knowledge and transcribing explicit information while awareness networks require displaying and monitoring one’s activities. Expertise networks prescribe new contacts or remaking old contacts, while awareness networks emerge by observing coworkers’ activities. Here, an example can illustrate how the concepts blend in the middle of the spectrum. Namely, our own study of awareness networks revealed one practice wherein managers encouraged the use of a discussion database (Section 5.2). The objective was to help identify the right people for answering certain questions. Though this concept was a weak component (managers had to enforce its use) of maintaining awareness, it is more similar to expertise finding than awareness.

Finally, a key consideration for both expertise networks and awareness networks is their use for *synthesis*, such as building support tools, or for *analysis*, such as drawing attention to an overarching phenomenon. For instance, the paper by Nardi and colleagues is an example of using the concept of intensional networks for analytical purposes [39]. At the same time, Nardi and colleagues also developed a tool for supporting intensional networks

[40]. Zhang and colleagues used the concept of expertise networks to analyze on-line, question-answer forums and then synthesized *expertise ranking algorithms* [57]. Our present paper is a good example of the application of the concept of awareness networks as an analytical framework. We hope to see other researchers exploring this concept as a design feature of collaborative tools.

As discussed above, awareness networks and expertise networks differ in a number of dimensions, though in any one dimension, a degree of similarity can arise. Nonetheless, maintaining the distinction allows us to better understand how these networks operate, where and when they overlap in the work of software development. From an analytical point of view, we argue it is beneficial to maintain the conceptual distinction between these terms. To exemplify this point, we draw the reader's attention to the example given above wherein managers encouraged the use of a discussion database. As we noted, this example comes the closest in terms of a blend of an expertise network perspective and an awareness network perspective. We also commented that this discussion database was a weak component of our subjects' awareness practices. Were we taking an expertise network perspective, we most certainly would have identified the discussion database as a key component and its lack of use as problematic. On the other hand, the PR work (section 4.3) is a perfect example of work practice towards the identification of awareness networks, but that would not be a good fit for expertise networks. The distinction between these networks is also beneficial for the purposes of synthesis, i.e., tool designers can more adequately inform the design of collaborative systems to support these different kinds of networks. For instance, since the scale and timeframe of awareness and expertise networks is different, the mechanisms to access, represent and visualize the information portrayed in such networks should be different as well.

## 8 CONCLUDING REMARKS

In the field of Computer-Supported Cooperative Work, the term awareness is used to describe a range of work practices by which social actors coordinate their work through (i) the display of their actions to their colleagues, and (ii) the monitoring of actions from their colleagues. Recently, this concept has been explored by software engineering researchers in the design of collaborative software development tools.

Most empirical studies related to awareness focus on the identification of these coordinative practices and assume settings in which the social actors who display and monitor actions do not change often. However, the practices of displaying and monitoring actions associated with awareness are useful only to the extent that social actors *know who they should monitor and to whom they should display their actions*. In collocated settings, this information is intrinsic. However, there are settings where this information is not as clear, e.g., distributed software projects. Previous studies have largely overlooked the identification of these actors. Accordingly, this paper focuses on the soft-

ware developers' work practices necessary to identify the list of actors whose actions should be monitored and to whom actions should be displayed. We call this set of actors the *awareness network*. In shifting the focus, it is possible to observe a myriad of such practices, how they are influenced by the work setting (organization, software architecture, etc), the problems that arise when this identification is problematic, and, finally, software developers' concern with the management of these networks.

We have drawn our results from empirical data from three software development teams that were observed and interviewed. Results of our analysis suggest that the awareness network of a software developer is fluid (it changes during the course of software development work) and is influenced by three main factors: the organizational setting (e.g., the reuse program in the BSC corporation), the software architecture, and, finally, the recency of the project. Finally, we observed that software developers try to manage their awareness networks to be able to handle the impact of interdependent actions.

These results can lead to important contributions for collaborative Software Engineering. For instance, collaborative tools (like Ariadne [13, 53], CollabCVS [32], Fast-DASH [4], Jazz [10], and Palantir [43]) and theoretical approaches (like socio-technical congruence [7, 9]) that leverage software architecture to provide awareness have provided some promising results. However, the problem with these approaches is that, in practice, the architecture is not fully disclosed to or fully understood by all developers. The fluidity of the awareness networks suggest that these same tools and approaches need to be flexible to easily adapt to a software developer's current context, including task, and software process phase. Similarly, most approaches and tools do not properly describe the organizational context where they would be adequate leading to misinterpretation and generalization of results.

Our results indicate that the ways people identify the colleagues that they need to be aware of is a necessary and integral aspect of collaborative work. Being aware, and identifying who one needs to be aware of, is a software developer's ongoing achievement and needs to be properly studied and supported by software tools.

## ACKNOWLEDGMENTS

The first author was supported by the Brazilian Government under grant CNPq 473220/2008-3 and by the Fundação de Amparo à Pesquisa do Estado do Pará (FAPESPA) through "Edital Universal N° 003/2008". Both authors received support for this work from the U.S. National Science Foundation under grant 0534775 and through an IBM Eclipse Technology Exchange grant. The second author also received funding from the National Science Foundation under grant 0205724. We also wish to thank our technical editor, Sandra Rush, for fast editing of earlier versions of this work. Any remaining errors are our own. We wish to thank Paul Dourish, Stewart Sutton, Matthew Bietz, our other colleagues, the guest editors, and anonymous reviewers for their helpful feedback. Finally, very importantly, we wish to thank the Alpha, Beta, and Gamma developers for allowing the data collection

and providing feedback on their work.

## REFERENCES

- [1] Ackerman, M.S., et al., Who's There? The Knowledge Mapping Approximation Project, in *Sharing Expertise: Beyond Knowledge Management*, M.S. Ackerman, V. Pipek, and V. Wulf, Editors. 2002, MIT Press.
- [2] Amrit, C. and J. van Hillegersberg, Detecting Coordination Problems in Collaborative Software Development Environments. *Information Systems Management*, 2008. 25: p. 57-70.
- [3] Bentley, R., et al. Ethnographically-informed systems design for air traffic control. in *ACM Conference on Computer Supported Cooperative Work*. 1992. Toronto, Ontario, Canada: ACM Press.
- [4] Biehl, J.T., et al., FASTDash: a visual dashboard for fostering awareness in software teams, in *Proceedings of the SIGCHI conference on Human factors in computing systems2007*, ACM: San Jose, California, USA. p. 1313-1322.
- [5] Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering1974*: Addison-Wesley. 336.
- [6] Buschmann, F., et al., *Pattern-Oriented Software Architecture: A System of Patterns1996*, Chichester, West Sussex, UK: Wiley.
- [7] Cataldo, M., J.D. Herbsleb, and K.M. Carley, Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity, in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement2008*, ACM: Kaiserslautern, Germany.
- [8] Cataldo, M., et al., Software Dependencies, Work Dependencies and Their Impact on Failures (forthcoming). *IEEE Transactions on Software Engineering*, 2009.
- [9] Cataldo, M., et al., Identification of Coordination Requirements: implications for the Design of Collaboration and Awareness Tools, in *20th Conference on Computer Supported Cooperative Work2006*, ACM Press: Banff, Alberta, Canada.
- [10] Cheng, L.-T., et al., Building Collaboration into IDEs. Edit -> Compile -> Run -> Debug -> Collaborate?, in *ACM Queue2003*. p. 40-50.
- [11] Conway, M.E., How Do Committees invent? *Datamation*, 1968. 14(4): p. 28-31.
- [12] Curtis, B., H. Krasner, and N. Iscoe, A field study of the software design process for large systems. *Communications of the ACM*, 1988. 31(11): p. 1268-1287.
- [13] de Souza, C.R.B., et al. From Technical Dependencies to Social Dependencies. in *Workshop on Social Networks for Design and Analysis: Using Network Information in CSCW*. 2004. Chicago, IL, USA.
- [14] de Souza, C.R.B. and D. Redmiles, The Awareness Network: Should I Display my Actions to Whom? And, whose actions should I monitor?, in *European Conference on Computer-Supported Cooperative Work2007*, Springer: Limerick, Ireland. p. 99-117.
- [15] de Souza, C.R.B. and D. Redmiles, An empirical study of software developers' management of dependencies and changes, in *Proceedings of the 30th international conference on Software engineering2008*, ACM: Leipzig, Germany. p. 241-250.
- [16] de Souza, C.R.B. and D. Redmiles, On the Roles of APIs in the Coordination of Collaborative Software Development. *Journal of Computer Supported Cooperative Work*, 2009. 18(5-6): p. 445-475.
- [17] de Souza, C.R.B., et al., Guest Editors' Introduction: Cooperative and Human Aspects of Software Engineering, in *IEEE Software2009*, IEEE Press. p. 17-19.
- [18] Dourish, P. and V. Bellotti. Awareness and Coordination in Shared Workspaces. in *Conference on Computer-Supported Cooperative Work (CSCW '92)*. 1992. Toronto, Ontario, Canada: ACM Press.
- [19] Ehrlich, K., Locating Expertise: Design Issues for an Expertise Locator System, in *Sharing Expertise: Beyond Knowledge Management*, M.S. Ackerman, V. Pipek, and V. Wulf, Editors. 2002, MIT Press.
- [20] Ehrlich, K. and K. Chang. Leveraging expertise in global software teams: Going outside boundaries. in *International Conference on Global Software Engineering*. 2006. Florianópolis, Brazil: IEEE Press.
- [21] Estublier, J. and S. Garcia, Process model and awareness in SCM, in *Proceedings of the 12th international workshop on Software configuration management2005*, ACM: Lisbon, Portugal. p. 59-74.
- [22] Fuggetta, A. Software Processes: A Roadmap. in *Future of Software Engineering*. 2000. Limerick, Ireland.
- [23] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series1995, Reading, MA: Addison-Wesley.
- [24] Grinter, R.E., *Recomposition: Coordinating a Web of Software Dependencies*. *Journal of Computer Supported Cooperative Work*, 2003. 12(3): p. 297-327.
- [25] Grudin, J., *Computer-Supported Cooperative Work: History and Focus*, in *IEEE Computer1994*, IEEE Press. p. 19-26.
- [26] Gutwin, C. and S. Greenberg, A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Journal of Computer Supported Cooperative Work*, 2002. 11(3-4): p. 411-466.
- [27] Harper, R., J. Hughes, and D. Shapiro, Working in harmony: An examination of computer technology in air traffic control, in *European Conference on Computer Supported Cooperative Work1989*, Springer/Kluwer: Gatwick, London.
- [28] Heath, C., et al. Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room. in *European Conference on Computer-Supported Cooperative Work*. 1993. Milan, Italy: Springer/Kluwer.
- [29] Heath, C. and P. Luff, Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms. *Journal of Computer Supported Cooperative Work*, 1992. 1(1-2): p. 69-94.
- [30] Heath, C. and P. Luff, *Technology in Action2000*, Cambridge: Cambridge University Press. 269.

- [31] Heath, C., et al., Configuring Awareness. *Journal of Computer Supported Cooperative Work*, 2002. 11(3-4): p. 317-347.
- [32] Hedge, R. and P. Dewan, Connecting Programming Environments to Support ad-Hoc Collaboration, in *Automated Software Engineering 2008*, IEEE Press: L'Aquila, Italy. p. 178-87.
- [33] Herbsleb, J.D., et al. An Empirical Study of Global Software Development: Distance and Speed. in *International Conference on Software Engineering*. 2001. Toronto, Canada: IEEE Press.
- [34] Jorgensen, D.L., *Participant Observation: A Methodology for Human Studies* 1989, Thousand Oaks, CA: SAGE publications.
- [35] Kantor, M. and D. Redmiles. Creating an Infrastructure for Ubiquitous Awareness. in *Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001)*. 2001. Tokyo, Japan.
- [36] Löwstrand, L. Being Selectively Aware with the Khronika System. in *European Conference on Computer Supported Cooperative Work*. 1991. Amsterdam, The Netherlands: Springer/Kluwers.
- [37] Malone, T.W. and K. Crowston, *The Interdisciplinary Study of Coordination*. *ACM Computing Surveys*, 1994. 26(1): p. 87-119.
- [38] McCracken, G., *The Long Interview* 1988, Thousand Oaks, CA: SAGE Publications.
- [39] Nardi, B., S. Whittaker, and H. Schwarz, *NetWORKers and their Activity in Intensional Networks*. *Journal of Computer Supported Cooperative Work*, 2002. 11(1-2): p. 205-242.
- [40] Nardi, B.A., et al., Integrating communication and information through ContactMap. *Communications of the ACM*, 2002. 45(4): p. 89-95.
- [41] Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 1972. 15(12): p. 1053-1058.
- [42] Perry, D.E., H.P. Siy, and L.G. Votta, Parallel Changes in Large-Scale Software Development: An Observational Case Study. *ACM Transactions on Software Engineering and Methodology*, 2001. 10(3): p. 308-337.
- [43] Sarma, A., Z. Noroozi, and A. van der Hoek. Palantír: Raising Awareness among Configuration Management Workspaces. in *Twenty-fifth International Conference on Software Engineering*. 2003. Portland, Oregon.
- [44] Schmidt, K., The Problem with 'Awareness' - Introductory Remarks on 'Awareness in CSCW'. *Journal of Computer Supported Cooperative Work*, 2002. 11(3-4): p. 285-298.
- [45] Schmidt, K., Divided by a common acronym: On the fragmentation of CSCW, in *Proceedings of the European conference on Computer Supported Cooperative Work 2009*, Springer London: Vienna, Austria. p. 223-242.
- [46] Schmidt, K. and C. Simone, Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Journal of Computer Supported Cooperative Work*, 1996. 5(2-3): p. 155-200.
- [47] Sommerville, I., *Software Engineering*. 6th ed 2000, Boston, MA: Addison-Wesley Publishing Co.
- [48] Sosa, M.E., S.D. Eppinger, and C.M. Rowles, The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, 2004. 50(12): p. 1674-1689.
- [49] Staudenmayer, N.A., *Managing Multiple Interdependencies in Large Scale Software Development Projects*, in *Sloan School of Management 1997*, Massachusetts Institute of Technology: Cambridge, MA, USA.
- [50] Suchman, L., *Centers of Coordination: A Case and Some Themes*, in *Discourse, Tools, and Reasoning*, L. Resnick, R. Saljo, and C. Pontecorvo, Editors. 1997, Springer-Verlag. p. 41-62.
- [51] Teasley, S., et al. How Does Radical Collocation Help a Team Succeed? in *Conference on Computer Supported Cooperative Work*. 2000. Philadelphia, PA, USA: ACM Press.
- [52] Thompson, J.D., *Organizations in Action: Social Sciences of Administrative Theory* 1967, New Brunswick: Transaction Publishers.
- [53] Trainer, E., et al. Bridging the Gap between Technical and Social Dependencies with Ariadne. in *Eclipse Technology Exchange*. 2005. San Diego, CA.
- [54] Treude, C. and M.-A. Storey, Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 12010*, ACM: Cape Town, South Africa. p. 365-374.
- [55] Valletto, G., et al., Using Software Repositories to Investigate Socio-technical Congruence in Development Projects, in *Workshop on Mining Software Repositories 2007*, ACM Press: Minneapolis.
- [56] Van de Ven, A.H., A.L. Delbecq, and R. Koenig Jr., Determinants of Coordination Modes within Organizations. *American Sociological Review*, 1976. 41(2): p. 322-338.
- [57] Zhang, J., M.S. Ackerman, and L. Adamic, Expertise networks in online communities: structure and algorithms, in *Proceedings of the 16th international conference on World Wide Web 2007*, ACM: Banff, Alberta, Canada. p. 221-230.

**Cleidson R. B. de Souza** received his BS in Computer Science from the Federal University of Pará, Brazil in 1996 and his MS in Computer Science from State University of Campinas, Brazil in 1998. He received his Ph.D. in Information and Computer Science in 2005 from University of California, Irvine. From 1998 to 2010 he worked as an Associate Professor at the Federal University of Pará, Brazil. Beginning 2010, he became a Research Assistant at IBM Research – Brazil working with collaboration technologies to study service science and software engineering. His research interests are in the intersection between software engineering and computer-supported cooperative work, and service science.

**David F. Redmiles** received his BS in Mathematics and Computer Science in 1980 and his MS in Computer Science in 1982 from the American University, Washington, D.C. He received his PhD in Computer Science in 1992 from the University of Colorado, Boulder. From 1979 to 1987, he worked at the National Institute of Standards and Technology (formerly the National Bureau of Standards), Gaithersburg. From 1992 to 1994, he did postdoctoral work at the University of Colorado, Boulder. Since 1994, he has been at the University of California, Irvine, where he is currently Professor and Chair of the Department of Informatics in the Donald Bren School of Information and Computer Sciences. He has a background in software engineering, human-computer interaction, and computer-supported cooperative work. For the past decade, he has been re-

searching collaborative software engineering. He is a member of the IEEE.