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

Cleidson R. B. de Souza
Faculdade de Computação, Universidade Federal do Pará,
Belém, PA, Brasil
cdesouza@ufpa.br

David Redmiles
Department of Informatics, University of California, Irvine
Irvine, CA, USA
redmiles@ics.uci.edu

Abstract. The concept of awareness has come to play a central role in CSCW research. The coordinative practices of displaying and monitoring have received attention and have led to different venues of research, from computational tool support, such as media spaces and event propagation mechanisms, to ethnographic studies of work. However, these studies have overlooked a different aspect of awareness practices: the identification of the *social actors who should be monitored and the actors to whom their actions should be displayed*. The focus of this paper is on how social actors answer the following questions: *to whom* should I display my actions? And, *whose* actions should I monitor? Ethnographic data from two software development teams are used to answer these questions. In addition, we illustrate how software developers' work practices are influenced by three different factors: the organizational setting, the age of the project, and the software architecture.

Introduction

Schmidt (2002) discusses some important findings about the concept of awareness recognized by the CSCW community. These findings are based on seminal studies of work practice (Harper, Hughes et al. 1989; Heath and Luff 1992; Heath, Jirotka et al. 1993), and they conceptualize awareness as a range of coordinative practices performed by competent actors to accomplish their work (Heath, Svensson et al. 2002). 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 *display* their actions in such a way that others can easily monitor them[1]. 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.

The practices by which social actors became aware of their colleagues' work usually have been associated with actors' achievements—"hidden" results of work arrangements—and not viewed as the result of deliberate, explicit actions (Schmidt 2002). However, this is not the case. In fact, according to Schmidt, social actors deftly choose the degree of obtrusiveness of their actions:

> "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."

Despite the undeniable importance of these findings, one aspect has not received enough analytical attention by the CSCW community: the identification of social actors involved in the coordinative practices of awareness, that is, *how social actors identify the colleagues who should be monitored and those colleagues to whom their actions should be displayed*. We argue that a change of focus is required: instead of focusing on the coordinative practices, one should focus on how social actors answer the following questions: *to whom* should I display my actions? And, *whose* actions should I monitor? It is also necessary to understand how the organizational setting facilitates the identification of these two sets of actors.

These questions have been looked at from a technological point of view in event notification servers (Lövstrand 1991; Fitzpatrick, Kaplan et al. 2002), usually through subscriptions that allow one to define the notifications to receive.

---

1 Implicit in this discussion is the notion of interdependent activities, i.e., displaying and monitoring are especially relevant because the outcome of one's action can affect others' actions (Malone and Crowston 1994; Schmidt 2000).

Empirical studies, however, have not focused on these aspects, partly because the studies of work practice that helped to establish the concept of awareness used the perspectives of ethnomethodology and conversation analysis (Garfinkel 1967). Studies using these perspectives focused on the organization of the work in "small time frames"; consequently, social actors did not change. The settings studied (control rooms, newsrooms, trading rooms, etc.) required individuals to monitor their colleagues' immediate actions at the same time they were engaged in other activities (Heath, Svensson et al. 2002). Note that this is not a criticism of these sociological perspectives; rather, it is an observation that this focus has led CSCW researchers to overlook other aspects of awareness, as discussed further in this paper.

We can thus describe the focus of this paper as the identification of the "awareness network"—the network of actors whose actions need to be monitored and those to whom one needs to make one's own actions visible. Through the presentation of ethnographic data from two software development teams, we illustrate how software developers identify their awareness networks, the size and fluidity of these networks, and how these aspects influence the practices by which they become aware of the actions of their colleagues. We also discuss how organizational settings facilitate or hinder the identification and maintenance of awareness networks. In this regard, this paper briefly illustrates how software developers' knowledge about the software architecture is used to guarantee a smooth flow of work.

The remainder of this paper is organized as follows. The next section describes the two research sites studied, Alpha and Beta, as well as the methods used to collect and analyze data from these sites. Next, the ethnographic data of the Beta and Alpha teams is presented, and a discussion follows in the subsequent sections. Finally, the last section presents the final comments and future work.

Research Site and Methods

We conducted two qualitative studies at different large software development organizations. The first field study was conducted during summer 2002, and the second one was performed during summer 2003. We adopted observation (Jorgensen 1989) and semi-structured interviews (McCracken 1988) for data collection. The role of the software architecture in the work practices was evident during the data collection; therefore, we explicitly tried to collect information about this aspect. Data analysis was conducted by using grounded theory techniques (Strauss and Corbin 1998). Details about each team as well as the methods used are described next.

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 options 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.

The Alpha group adopts a formal software development process (Fuggetta 2000) 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 occur in the software. Another part of the process prescribes that, after checking-in files in 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. He 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.

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. 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 (Gamma, Helm et al. 1995), 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 (Buschmann, Meunier et al. 1996). 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 (de Souza, Redmiles et al. 2004).

Regarding data collection in this field study, we also adopted non-participant observation (Jorgensen 1989) and semi-structured interviews (McCracken 1988), which involved the first author spending 11 weeks at the field site. Among other documents, meeting invitations, 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 using grounded theory (Strauss and Corbin 1998) to understand the role of APIs in the coordination of Beta developers, as reported elsewhere (de Souza, Redmiles et al. 2004).

Data Analysis

After the second data collection, datasets from the two different organizations and projects were integrated into a software tool for qualitative data analysis, MaxQDA2. After that, the data collected was analyzed by using grounded theory (Strauss and Corbin 1998) with the purpose of identifying a framework to explain the results observed in both field studies. Interviews and field notes were coded to identify categories that were later interconnected with other categories.

The following sections describe the work practices of the Alpha and Beta teams, how their developers identify their awareness networks, and the organizational factors that influence the identification practices. More details can be found in the dissertation of the first author (de Souza 2005).

The Awareness Network in the Alpha Team

The Task Assignment

For accountability purposes, all changes in the Alpha software need to be associated with a problem report. A PR describes the changes in the code, the reason for the changes (bug fixing, enhancement, etc.), and who made the changes, among other pieces of information. An Alpha developer is usually delegated new tasks by being assigned to work with one or more PRs. These PRs are reported by other team members, who are 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 it 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 allocate developers to work on PRs that affect "their" respective processes. However, it is not unusual to find developers working in different processes[2]. In this case, Alpha developers need to identify and contact the process owner to find out whether there is a problem in the process[3]. 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 software development process), whether their changes in the process are going to impact the work of these process owners.

Finding Out 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:

---

2 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 understanding it, than to assign it to a different developer at that point.

3 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 (Curtis, Krasner et al. 1988). 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.

> "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."

The quote above illustrates how new members have difficulty in identifying who to contact for help, that is, their awareness network is unknown. This also illustrates how software developers use an organizational guideline (broadcast emails for each check-in) to handle this problem.

The Code Reviews

The Alpha software development process also prescribes the usage of code reviews to be performed by process leaders whose processes are affected by the changes in the code. This means that after a developer is done with changes in the Alpha software, he or she needs to request a code review in that code. 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. Furthermore, developers' changes can be reviewed as many times as required until they are allowed to be checked-in. More specifically, according to the Alpha development process:

> "If the appropriate CSCI Lead(s) decide that the Developer's code changes are not sufficient for the task, then the Lead(s) communicate with the Developer, then steps 6.1.13 through 6.1.16 are repeated until the CSCI Lead(s) decide that no further changes are required to accomplish the task." *[Alpha Software Development Process Description]*

As discussed in the previous section, developers need to identify process leaders in order to request code reviews; they therefore need to identify their awareness networks. Again, this is not a problem for most developers, but newcomers use emails to obtain that the information.

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[4]. The information about the impact on the V&V staff is recorded 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 to 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.

Sending Email

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 unmaintainable. … *yeah, every process talks to every other one*…" [emphasis added]

We discuss later 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.

Using Email

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 in their respective workspaces. Note that if a developer, John, is engaged in parallel development with another developer, Mary, and Mary already checked-in her changes in the main branch

---

4 Process leads also use information about changes in the code to perform code reviews.

before John did, John will necessarily have received an email from Mary about her check-in's. 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. 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 (CM) 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. What I will probably try to do is discard my modifications and/or I'll save my modifications and then, right now I'll see if I can put myself on top of it because at that point there's stuff supposedly already committed so there's nothing I can do except build on top of them."

To avoid back merges without avoiding parallel development, Alpha developers perform "partial check-in's." 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-in's to avoid being affected by other developer's changes in the same files because these changes can generate additional work for the developers.

The Awareness Network in the Beta Team

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 to 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 (Conway 1968; Parnas 1972).

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 to reduce software development costs. However, due to the size and geographical distribution of the organization, this 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."

Finding Out Who to Contact

In order to identify who they need to contact, developers adopt different approaches. First, they rely on their personal social networks. 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.

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 infra-structure 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, the manager of the client team decided to "lend" this developer to the external team[5]. By doing this, the manager, to some extent, could guarantee that the services he needed would be implemented. This approach provides another advantage: managers would guarantee the stability of part of their awareness network.

Identifying who 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 that [the discussion database] is the place to go when you 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.

---

5 In software engineering, artifact dependencies (such as the ones that exist among the components of a software system), often imply dependencies among software developers (Grinter 2003; de Souza, Froehlich et al. 2005).

> 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.

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 with client developers changes 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, according to Beta-Developer-15:

> "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]

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 related a 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." [6]

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 because 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."

Discussion

Before proceeding with the analysis of the data, it is important to clearly establish the differences and similarities between the Alpha and Beta teams, as presented in Table 1.

---

6 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.

Table 1 - Alpha and Beta Teams

	Alpha	Beta
Project duration	9 years	9 months
Team size	34	57
Sub teams	2	5
Formal software development process?	Yes	Partially. Only regarding the APIs
Software architecture	Non-modular and not-documented	Modular, defined through a reference architecture
Division of labor	Based on PR	Based on APIs
Interaction with other teams?	Not necessary	Often, due to the reuse program

The Identification of the Awareness Network

The empirical data presented in the previous section stresses the importance of the proper identification of a software developer's awareness network. Observations from the Beta team suggest that when the awareness network is misidentified, the collaborative endeavor is severely damaged; most of the problems faced by Beta developers (delays in their 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. They have to use several approaches ranging from activations of their personal networks to discussion databases. The situation is better in the Alpha team: due to the duration of the project, most developers already know their colleagues' expertise. The only exception is newcomers, who do not know this information and use email to identify the process leaders.

The identification of the awareness network is difficult because awareness networks are fluid. That is, they easily change components (the software developers) 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 (Sommerville 2000). 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 becomes 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 and 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 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: 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 services are 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 unknown. As the awareness network expands, software developers' work practices need to change as well to accommodate this situation.

Note, however, that the fluidity of the awareness networks in the Alpha and Beta teams is different: whereas it changes somewhat rapidly during the course of work on a PR for Alpha developers, it changes slowly in the Beta team. Furthermore, changes in the Alpha developers' network are temporary (they last only while the PR work lasts), whereas in the Beta teams they are permanent, at least until the next change in assignments.

At this point, it is important to compare the concept of awareness network with the one of intensional networks (Nardi, Whittaker et al. 2002). The former concept calls attention to the set of social actors that one needs (i) to be aware of, and (ii) to be made aware of, as well as the work required to identify those actors. Intensional networks focus on the work necessary to create, maintain, and activate personal social networks. To simplify, one could argue that the latter is a prerequisite for the former.

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 isn't 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.

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, 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. 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. Alpha software developers also use their knowledge about the software architecture (some processes are often changed in parallel) to avoid needing to monitor other software developers. 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 (Conway 1968; Parnas 1972; Grinter 2003; de Souza, Redmiles et al. 2004; de Souza 2005). What is important here is finding out *how* software developers make use of that information to facilitate their work.

Concluding Remarks

The term "awareness" is used to describe a range of practices by which social actors coordinate their work through the display of their actions to their colleagues and the monitoring of actions from their colleagues. Most empirical studies 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. This happens because the seminal studies of awareness practices usually adopted perspectives (ethnomethodology or conversation analyss) that focused on actions in a small time frame. Furthermore, these studies focused on settings such as control rooms, newsrooms, and trading rooms, which have characteristics that make necessary for individuals to monitor each other's conduct on an ongoing basis while engaged in distinct but related activities (Heath, Svensson et al. 2002).

This paper departs from this view and takes a different approach. It focuses on the software developers' work practices necessary to accomplish their work over a somewhat extended period of time. By doing that, it is possible to observe how these practices are influenced by the organizational setting, and, more important, how developers' coordinative practices require proper identification and maintenance of the list of actors whose actions should be monitored and to whom actions should be displayed. We call this list the "awareness network." The practices of displaying and monitoring are useful only to the extent that social actors do know who they should monitor and to whom they should display their actions. Previous studies had largely overlooked this aspect.

We have drawn our results from empirical data from two software development teams that were observed and interviewed. Their data were analyzed by using grounded theory techniques (Strauss and Corbin 1998). Our results suggest that the awareness network of a software developer is fluid (it changes during the course of work) and is influenced by three main factors: the organizational setting (the reuse program in the BSC corporation), the software architecture, and, finally, the recency of the project. In addition, software developers even try to manage their awareness networks to be able to handle the impact of interdependent actions.

Acknowledgments

This research was supported by the Brazilian Government under grants CAPES BEX 1312/99-5 and CNPq 479206/2006-6, by the U.S. National Science Foundation under grants 0534775 and 0205724, and by an IBM Eclipse Technology Exchange grant.

References

Buschmann, F., R. Meunier, et al. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, West Sussex, UK.

Conway, M. E. (1968). 'How Do Committees Invent?' *Datamation*, vol. 14, no. 4, pp. 28-31.

Curtis, B., H. Krasner, et al. (1988). 'A Field Study of the Software Design Process for Large Systems.' *Communications of the ACM*, vol. 31, no. 11, pp. 1268-1287.

de Souza, C. R. B. (2005). On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support. Department of Informatics, Donald Bren School of Information and Computer Sciences. University of California, Irvine. Ph.D. dissertation, p. 186.

de Souza, C. R. B., J. Froehlich, et al. (2005). Seeking the Source: Software Source Code as a Social and Technical Artifact. ACM Conference on Supporting Group Work, Sanibel Island, FL, November 06-09, 2005, pp. 197-206.

de Souza, C. R. B., D. Redmiles, et al. (2004). Sometimes You Need to See Through Walls—A Field Study of Application Programming Interfaces. Conference on Computer-Supported Cooperative Work, Chicago, IL, November 6-10, 2004, pp. 63-71.

Fitzpatrick, G., S. Kaplan, et al. (2002). 'Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections.' *Journal of Computer Supported Cooperative Work*, vol. 11, no. 3-4, September, 2002, pp. 299-316.

Fuggetta, A. (2000). Software Processes: A Roadmap. Future of Software Engineering, Limerick, Ireland.

Gamma, E., R. Helm, et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Garfinkel, H. (1967). *Studies in Ethnomethodology*. Prentice-Hall, Englewood Cliffs, NJ.

Grinter, R. E. (2003). 'Recomposition: Coordinating a Web of Software Dependencies.' *Journal of Computer Supported Cooperative Work*, vol. 12, no. 3, 2003, pp. 297-327.

Harper, R., J. Hughes, et al. (1989). Working in Harmony: An Examination of Computer Technology in Air Traffic Control. European Conference on Computer Supported Coooperative Work. Springer/Kluwer, Gatwick, London.

Heath, C., M. Jirotka, et al. (1993). Unpacking Collaboration: The Interactional Organisation of Trading in a City Dealing Room. European Conference on Computer-Supported Cooperative Work, Milan, Italy, September 13-17, pp. 155-170.

Heath, C. and P. Luff (1992). 'Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms.' *Journal of Computer Supported Cooperative Work*, vol. 1, no. 1-2, 1992, pp. 69-94.

Heath, C., M. S. Svensson, et al. (2002). 'Configuring Awareness.' *Journal of Computer Supported Cooperative Work*, vol. 11, no. 3-4, September 2002, pp. 317-347.

Jorgensen, D. L. (1989). *Participant Observation: A Methodology for Human Studies*. SAGE Publications, Thousand Oaks, CA.

Lövstrand, L. (1991). Being Selectively Aware with the Khronika System. European Conference on Computer Supported Cooperative Work, Amsterdam, The Netherlands, September 24-27, pp. 265-279.

Malone, T. W. and K. Crowston (1994). 'The Interdisciplinary Study of Coordination.' *ACM Computing Surveys*, vol. 26, no. 1,1994, pp. 87-119.

McCracken, G. (1988). *The Long Interview*. SAGE Publications, Thousand Oaks, CA.

Nardi, B., S. Whittaker, et al. (2002). 'NetWORKers and their Activity in Intensional Networks.' *Journal of Computer Supported Cooperative Work*, vol. 11, no. 1-2,2002, pp. 205-242.

Parnas, D. L. (1972). 'On the Criteria to be Used in Decomposing Systems into Modules.' *Communications of the ACM*, vol. 15, no. 12,1972, pp. 1053-1058.

Schmidt, K. (2000). The Critical Role of Workplace Studies in CSCW. In Workplace Studies: Recovering Work Practice and Informing System Design, P. Luff, J. Hindmarsh and C. Heath, Cambridge University Press, Cambridge, UK, pp. 141-149.

Schmidt, K. (2002). 'The Problem with 'Awareness'—Introductory Remarks on 'Awareness in CSCW'.' *Journal of Computer Supported Cooperative Work*, vol. 11, no. 3-4, September 2002, pp. 285-298.

Sommerville, I. (2000). *Software Engineering*. Addison-Wesley Publishing Co., Boston, MA.

Strauss, A. and J. Corbin (1998). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Thousand Oaks, CA.