A Group Awareness and collaboration in Distributed Software Development

Imran Ali Siddiqui, Manuj Darbari

Abstract—. Open-source software development projects manage to produce large, robust, complex, and successful systems. OSS are always collaborative and distributed in nature as well as difficulties are being occurred due to distance. However, there is a little spite of knowledge about management of collaboration by open-source team. In this paper we look how distributed developers maintain group awareness. We interviewed developers, read project communication, and looked at project artifacts from three successful open source projects. We found that distributed developers need to maintain awareness of one another, and that they maintain both a general awareness of the entire team and more detailed knowledge of people with whom they plan to work.

Collaborative software development presents a variety of coordination and communication problems, particularly when teams are geographically distributed. One reason for these problems is the difficulty of staying aware of others – keeping track of information about who is working on the project, who is active, and people have been working with which task. Current software development environments do not show much information about people, and developers often must use text-based tools to determine what is happening in the group. We have built a system that assists distributed developers in maintaining awareness of others. Although there are several sources of information, this awareness is maintained primarily through text-based communication these textual channels have several characteristics that help to support the maintenance of awareness, as long as developers are committed to reading the lists and to making their project communication public.

Keywords: Collaboration Issues, Group awareness, Project Analyzer, Mining Component

1 INTRODUCTION

OBject Software development is used to work in current scenario of this real-world. Where work happens in a distributed fashion. In this regard Open-source software (OSS) development projects have developed projects by programmers from many different parts of the world, who rarely meet face-to-face due to distance. Software projects are most often carried out in a collaborative fashion. The complexities of software and the interdependencies between modules mean that these projects present collaborators with several coordination and communication problems. When development teams are geographically distributed, these problems often become much more serious [10, 18, 19, and 31]. Even though projects are often organized to try and make modules independent of one another, dependencies cannot be totally removed [31]. As a result, situations can arise where team members duplicate work, overwrite changes, make incorrect assumptions about another person's intentions, or write code that adversely affects another part of the project [18]. These problems occur because of a lack of awareness about what is happening in other parts of the project. Most development tools and environments do not make it easy to maintain awareness of others' activities [18]. Current tools are focused around the artifacts of collaboration rather than people's activities (files in a repository rather than the actions people have taken with them).

An artifact-based approach is clearly necessary for certain types of work, but without better information about people, smooth collaboration becomes difficult. Awareness is a design concept that holds promise for significantly improving the usability of collaborative software development tools. In this paper, we look that how distributed developers maintain group awareness. And group awareness information includes knowledge about who is on the project, where in the code they are working, what they are doing, and what are they planning. This knowledge seems vital if distributed developers are to coordinate their efforts, smoothly add code, make changes that affect other modules, and avoid rework. We carried out a study of open source teams to determine whether developers need to stay aware of one another, what awareness information developers keep track of, and how they gather and maintain their knowledge. We interviewed many developers on different well-established OSS projects, examined email and chat archives, and analyzed project artifacts such as source-code repositories, web pages, and official project documentation. We were surprised by our results. First, we expected that projects would be set up to reduce awareness requirements, with each software module carefully partitioned and protected from others. However, we found that official partitioning is limited, and that developers can contribute to any part of the code – an organizational approach that increases awareness requirements. Second, we found that the developers were able to maintain a good general awareness of other developers and their activities, and were able to find more detailed information about people's activities when they needed to. However, we were surprised that the main mechanisms for maintaining group awareness were simple text communication tools – developer mailing lists and text chat. Since these tools are disconnected from the project

• *Imran Ali Siddiqui, Research Scholar, JJTU Univrsity, Jhunjhunu, Rajasthan, India, E-mail: imranas@technologist.com*

• *Manuj Darbari is working as Associate Professor with the Department of InfomationTechnology in Babu Banarsi Das National Institute of Technology and Management,Lucknow, India E-mail: manujuma@gmail.com*

artifacts, and because they require explicit effort, we expected them to provide only incidental awareness – but in all three projects they were the main source of information. When we looked more closely at the email and chat messages, we found that these text channels have a number of characteristics that are valuable for the provision and collection of awareness information. First, they are public, and so allow all the developers on the list to become peripheral participants in each others' conversations. By overhearing others and by seeing who is talking about what, developers can gather important group awareness information. Second, mailing lists allow people to find out who the experts are in an area, simply by initiating a discussion: because the messages go to the entire group, the 'right people' will identify themselves by joining the conversation.

These awareness mechanisms can only work if most of the discussion between developers happens on the public channels and we found that there are strong elements of organizational culture on these projects that do just this. In particular, there is a strong culture of 'making it public' where developers are willing to answer questions, discuss their plans, report on their actions, and argue design details, all on the mailing list. Our findings provide details of how one kind of real-world distributed group maintains awareness and manages coordination, and exposes some of the underlying mechanisms that allow developers to overcome the problems of distance. We were impressed that ordinary verbal communication could be so effective in supporting awareness and coordination – particularly when the discussions so often refer to work artifacts that are not represented in the communication system. Although not all work settings are similar to open source development, we believe that our findings can assist analysis of awareness in other distributed work situations, and that the principles of awareness on OSS projects can benefit other types of computer-supported distributed work. Our study also suggests that groupware designers should tread carefully when inventing tools for distributed software development.

Awareness in collaboration network

Awareness has received attention in the Computer-Supported Cooperative Work (CSCW) community; this knowledge has not been considered extensively in development settings. We believe that awareness is a design concept that holds promise for significantly improving the usability of collaborative software development tools. Collaboration is an important research area of software engineering – where teams are common and good communication and coordination are essential for success. We review issues of collaboration in distributed software development, the basics of group awareness, and the awareness requirements that we have determined from observations of open source projects.

Collaboration Issues in Software Development

Collaboration support is a basic part of distributed development where teams have long used version control, email, chat groups, code reviews, and internal documentation to coordinate activities and distribute information. but these solutions generally either represent the project at a very coarse granularity require considerable time and effort or depend on people's current availability. Researchers in software engineering have found a number of problems that still occur in group projects and distributed software development. They found that it is difficult to determine when two people are making changes to the same artifacts [31] and communicate with others across time zones and work schedules [19]. They find partners for closer collaboration or assistance on particular issues [25] and also determine who has expertisation or deep knowledge about the different parts of the project [29]. They analyze that benefit from the opportunistic and unplanned contact that occurs when developers are co-located, since there is little visibility of others' activities. As Herbsleb and Grinter [18] state, lack of awareness – "the inability to share the same environment and to see what is happening at the other site" is one of the major factors in these problems. These are relevant issues to find the good coordination in distribute system.

Group Awareness

In many group work situations, awareness of others provides information that is critical for smooth and effective collaboration. Group awareness is the understanding of who is working with you, what they are doing, and how your own actions interact with theirs [13]. Group awareness is useful for coordinating actions, managing coupling, discussing tasks, anticipating others' actions, and finding help [16]. The complexity and interdependency of software systems suggests that group awareness should be necessary for collaborative software development. Knowledge of developer activities, both past and present, has obvious value for project management, but developers also use this information for many other purposes – purposes that assist the overall cohesion and effectiveness of the team. For example, knowing the specific files and objects that another person has been working on can give a good indication of their higher-level tasks and intentions; knowing who has worked most often or most recently on a particular piece of code indicates who to talk to before starting further changes; and knowing who is currently active can provide opportunities for real-time assistance and collaboration.

In co-located situations, three mechanisms help people to maintain awareness: explicit communication, where people tell each other about their activities; consequential communication [27], in which watching another person work provides information as to their activities and plans; and feed through [12], where observation of changes to project artifacts indicates who has been doing what. Of these mechanisms, explicit communication is the most flexible, and previous research has looked at the ways that groups communicate over distance, through email, text chat, and instant messaging [23,28]. However, since intentional communication of awareness information also requires the most additional effort, many awareness systems attempt to support implicit mechanisms as well as communication. General approaches include providing visible embodiments of participants and visual representations of actions that allow people to watch each other work, and overview visualizations of artifacts that show feed through information. Although group awareness is often taken for granted in face-to-face work, it is difficult to maintain in distributed settings.

This is particularly true in software development other than access to the shared code repository, development environments and tools provide almost no information about people on the project. Although communication tools such as email lists and chat systems help to keep people informed on some projects, these text-based awareness mechanisms require considerable effort, and are not well integrated with information about the artifacts of the project. As a result, coordination problems are common in distributed settings, and collaboration suffers. A few research systems do show awareness information (26, 14), but it is not clear that these tools really provide the awareness information that is needed by developers.

Need of Awareness in Distributed Software Development

The need for awareness therefore depends on the degree to which developers must coordinate. The main benefits of group awareness on a distributed software project would be in simplifying communication and improving coordination of activity. Software systems involve dependencies and linkages that require knowledge of others' activities. These dependencies can cause problems when development teams are distributed [7, 5]. There is a question that have open source projects managed to reduce these dependencies – simply based on the fact that they do manage to produce successful software? There are two ways that these dependencies can be reduced - by reducing the number of developers, or by strongly partitioning the code. The effects of increasing the number of developers has been studied before: Brooks' Law states that "the complexity and communication costs of a project rise with the square of the number of developers". Raymond [8] suggests that OSS projects avoid this explosion of connections by having only a small set of core developers, with a larger 'halo' of people whose activity is limited. Raymond discusses projects with one to three core developers, where awareness can likely be easily maintained through verbal communication; at the higher end, Mockus and colleagues [3] suggest that a core of ten to fifteen developers is the maximum that can be handled without the need to subdivide into separate subprojects. This number is large enough that the maintenance of awareness would not be simple. Responsibility is a strange concept in a collaborative volunteer project. With most things there are several people who know their stuff, so there's no clear concept of responsibility. The exception is of course where someone's name is down against something. For example, I put my name against the <abc> package as its maintainer, and so I am responsible for it. When I commit my new port, I will be responsible for that. On Apache http and Subversion, in contrast, official partitioning was almost nonexistent – on these projects, "all committers are responsible for all parts of the code" and in fact the traditions of both projects argue explicitly against partitioning. Part of the reason is likely that these projects are much smaller than Net BSD, but they have also found that ownership can cause as many coordination problems as it solves: Apache http developer A1: the paradigm is that all committers are responsible for all parts of the code. This is to lessen the impact of 'owned' modules. a few modules were 'owned' by a particular individual, but when that person left, the module rotted, those modules are detested by the general http community because they were never cleanly integrated and there was 'ownership' regarding that module that was never clearly relinquished. In general, we really try to avoid 'clear' ownership. It's been bad before when that's happened.

The summary statistics for http and Subversion reflect this attitude: the largest fraction that any developer contributes to any single file is about two thirds, and less than a third of the files are strongly associated with a single developer. The lack of clear partitioning reinforces the findings of Mockus and colleagues [3], who suggest that it is not simply the structure of a project that enables developers to coordinate their actions: Lack of clear ownership strongly suggests some other mechanism for coordinating contributions. It seems that rather than any single individual writing all the code for a given module, those in the core group have a sufficient level of mutual trust that they contribute code to various modules as needed. It is a matter of recognition of expertise than one of strictly enforced ability to make commits to partitions of the code base. This way of organizing projects through areas of expertise rather than through explicit partitioning does not remove awareness requirements; it actually increases them. When developers can work anywhere, they need to know who is active in the area, and who experts are. So group awareness becomes a critical component in successful coordination. When we asked developers what kinds of information about others that they tracked, they mentioned two types. First, developers maintain a broad awareness of who are the main people working on their project, and what their areas of expertise are. Second, when a developer wishes to do work in a particular area, they must gain more detailed knowledge about who are the people with experience in that part of the code.

Project Analyzer

Project Analyzer gathers information about project artifacts and developer's actions with those artifacts, and visualizes this awareness information either as a stand-alone tool or as a plug-in inside the Eclipse IDE. We have developed an awareness system called Project Analyzer to address some of the awareness issues that we have seen in distributed development projects. Project Analyzer consists of two main parts (i). Mining component (ii). Awareness visualizations.

Mining Component

The mining component analyzes a project's source code to produce facts for use by the Project Analyzer visualization displays. To gather developer activity information at a finer grain size than repository commits, a shadow CVS repository is maintained in Figure 1. User edits are auto-committed to the shadow repository as developers edit source code files with each auto-commit a new version of the file is stored in the shadow repository. The mining component analyzes the auto-committed versions against each other and the versions in the shared CVS (Concurrent Versioning System) repository to obtain user edit information that can be understood in terms of the project's software architecture. The mining component is composed of two fact extractors: the software architecture fact extractor and the user edit fact extractor. The software architecture fact extractor is run against the software repositoy to obtain entity relationship facts. Entity facts extracted include: package, class and method facts. Relationship facts extracted include: calls, contains, imports, implements and extends relationships. The software architecture facts are used by the visualization system to present the software structure. The user edit fact extractor is run against the shadow repository to obtain information about the methods a developer is changing. The user edit facts are used by the visualization to present developer activity information.

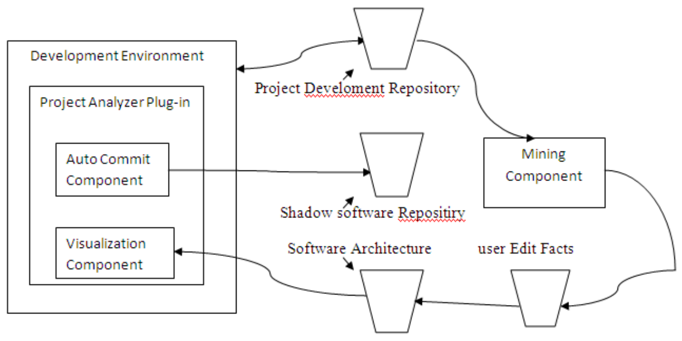

Figure 1: User edit fact extraction

The software architecture fact extractor is implemented in two stages and may either be run on the shadow repository or on the shared software repository in Figure 2. The first stage, the base fact extractor uniquely names the entities in the source code and extracts the facts of interest. This process is accomplished with a TXL [32] program using syntactic pattern matching [11]. The second stage, the reference analyzer, resolves references between software architecture entities. The reference analyzer extracts scope facts from the project source code and integrates them with the facts extracted in stage one. This process involves resolving the types of variables and return types of methods that are passed as arguments to method calls. The types of all the arguments are identified. Then scope, package, class, and method facts are analyzed to determine which package and class the method belongs to. To resolve calls to the Java library, the full Java API is first processed by the Project Analyzer mining component (this is only done once for all projects).

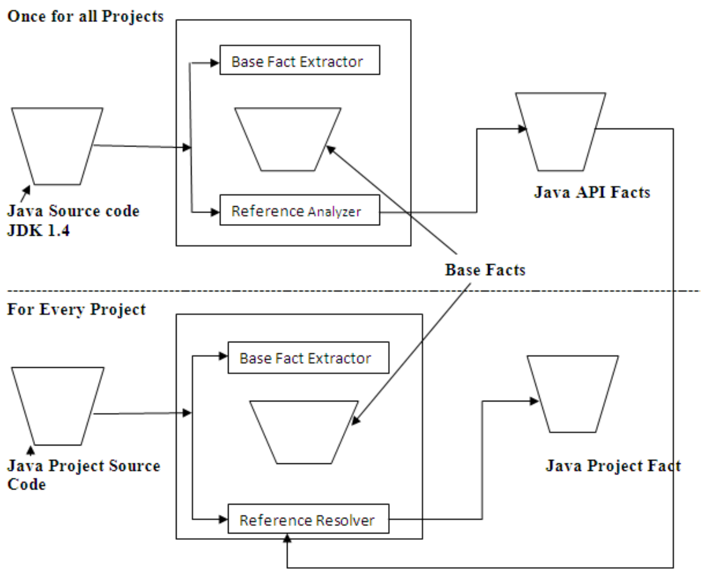

Figure 2: Software architecture fact extraction from Java projects

The user edit fact extractor (Figure 3) is implemented in three stages and is run against two versions of the project source code. The first stage splits the files into separate class and method snippets. The second stage compares and matches revisions of the code snippets. Initially, methods are matched based on their names. If a method match is not found at the method name level, methods are compared based on the percentage of lines of code that match between all methods. If a method's name is changed, a match based on percentage of similarity is still found between the two versions. When no match is found for a method from an earlier revision, the method is identified as having been added. When no match is found for a method from a later revision, the method is identified as having been removed. Facts about method additions and method removals are stored in the user edit fact base. Once the methods from each revision have been matched, a line diff is performed on each pair of methods. The diff algorithm gives us information about what lines have been added and removed from a method, and this information is stored in the user edit fact base. The complete fact base contains uniquely identified facts indicating all packages, classes, methods, variables, and relationships for a Java project and all user edits. These facts are used by the visualization component to show activity and proximity information. The time and space needed for fact extraction and fact base storage depends on the size of the code.

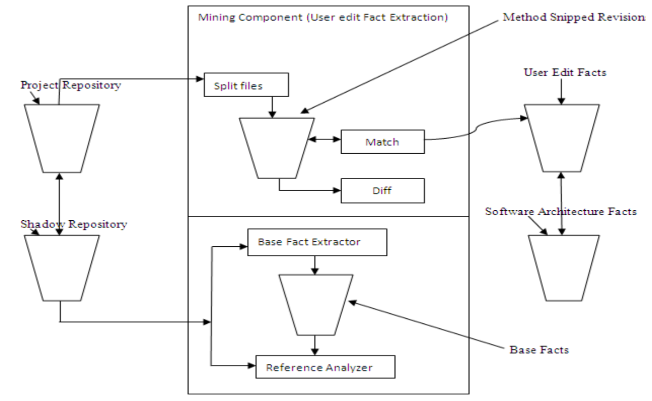

Figure 3: User edits fact extraction

Analyzer Report on activities and commits

Project Analyzer's activity awareness display visualizes team members' past and current activities on project artifacts. The goals of this display are to give collaborators an overview of who works on the project and provide a general sense of who works in what areas and they allow to changes to be tracked without much effort. The display uses the ideas of edit wear, interaction histories, and overviews. Edit wear is a concept introduced by Hill and colleagues [30]. Their overall motivation is the question of how computation can be used to improve "the reflective conversation with work materials" and the observation that most computational artifacts do not show any traces of the ways that they have been used, unlike objects in the real world.

The basic idea is to maintain and exploit object-centered interaction histories: record on computational object. The events that comprise their use and display useful graphical abstractions of the accrued histories as part of the objects themselves." [30] Hill and colleagues were primarily interested in an individual's reflection on their use of work artifacts, but there is obvious value for group awareness as well. In Project Analyzer, the artifacts are the files in a CVS repository (shadow or regular), and the interaction history is a record of all of the actions that a person undertakes with them (gathered unobtrusively by the fact extractor as people carry out their normal tasks). We take these interaction histories and visualize them on an overview representation of the entire project. Overviews provide a compact display of all the project artifacts, and allow information to be gathered at a glance. In addition, the overview representation can be overlaid with visual information about the interaction history or about changes to the artifacts. Although some tools such as CVS front-ends do limited visualization of the source tree. Our goal here is to collect much more information about interaction, and provide richer visualizations that will allow team members to quickly gather awareness information.

Related Work

A number of software engineering tools provide some degree of information about other members of the team (such as their identities or their assigned tasks), or provide facilities for team communication [10,14,24]. However, only a few systems combine information about people's activities with representations of the project artifacts. Two of them has done this very well as Augur [15] and TUKAN [25,26]. TUKAN is one of the first systems to explicitly address the question of awareness in software development. The basic representation used in a Smalltalk class browser, onto which awareness information is overlaid. In particular, the system shows the distance of other developers in 'software space,' using a software structure graph as the basis for calculating proximity. The main difference in our approach with Project Analyzer is in the use of an overview; where TUKAN [25] presents relevant information about others who may be encroaching on a developer's current location, Project Analyzer provides a general overview of the entire project. Augur is a system similar to Ball and Eick's [9], that presents line-based visualizations of source code along with other visual representations of the project.

The goal of Augur is to unify information about project activities with information about project artifacts; the system is designed to support both ongoing awareness and investigation into the details of project activity. Project Analyzer also uses the ideas of edit/read wear and combining activity and artifact information; the main difference between the two systems is that Augur is a large-scale system with many views and a highly detailed representation of the project, whereas Project Analyzer's visualization is designed only to support the two awareness questions seen in our work with existing projects ("who is who in general" and "who works in this area of the code"). In addition, Project Analyzer is based on a much finer temporal granularity of activity than is Augur, which uses repository commits as its source of activity information. We see Project Analyzer as more suited to day-to-day activities on a collaborative project, and specific investigations where developers wish to explore the history of the project in more detail.

Awareness Transportability

Our findings show how one kind of real-world distributed group maintains group awareness. Although this is only a small part of the overall story of how OSS teams overcome the problems of distance, we have been able to expose some of the information sources and mechanisms that allow these projects to stay coordinated in Summary of capabilities. Here we consider ways that our results can be used in the broader context: we look at whether the specific awareness mechanisms seen in the study could be used in other distributed work settings, whether there are underlying principles that can be applied more widely, and whether new tools could assist awareness on open-source projects.

Although simple text communication works well in these projects, and although text tools like MUDs have been successful in other work environments [12,16], it is not clear that email lists and chat systems are the answer for other distributed teams. Developers on open source projects are often 'the best of the best' in terms of technical skill and ability to get things and so it is possible that text-based awareness is feasible for them simply because they are very capable individuals. Also, open-source developers to some degree self-select for success in this environment: if a developer is not able to maintain adequate awareness and is not able to coordinate activity successfully, then because participation is voluntary, there is a good chance that they will not stay with the project. Finally, it appears that one of the primary motivations in open source communities is reputation among one's peers (rather than things like money or altruism) [23]. Although this is unlikely to be an explicit rating or ranking reputation is undoubtedly one reason why some of the more effortful parts of maintaining group awareness – reading the lists, writing good-quality responses, helping others – continue to be done by the majority of the community, and one reason why the communication forums sustain critical mass [5].

SUMMARY OF CAPABILITIES AND REQUIREMENTS FOR THE AWARENESS MECHANISMS

(a). Dynamic information sources:

Developer mailing lists: Overhearing is a primary mechanism; wide readership allows authors to reach 'the right people.' Requires additional communication effort, a strong culture of making things publicly and a critical mass of readers.

Text chat: Provides for ad-hoc communication and overhearing of informal and work-related discussions. Risk of removing communication from mailing lists; however, summaries can be posted back to the list.
Commits. Indicate people's activity levels and area of work. Can be time-consuming and tedious to read.

(b). On-demand awareness queries:

- Asking senior developers: Allows use of social networks to find other people. Requires explicit communication and an organizational culture that allows and promotes contact.
- 'Maintainer' field: Explicit indication of who is talking about changes. Effort is required to keep the information up to date; the project may not agree with code ownership.
- Code repository: Allows inspection of activity based on changes to project artifacts. Text-based displays mean that some information such as frequency of activity is difficult to see.
- Project documentation: Provides direct information about activities and areas of work. Must be kept up to date.
- Issue and bug trackers: Provide information about assignments, and show focused communication about each issue. Require explicit effort, and may remove communication from other lists.

There are many people outside of open source who are technically proficient, capable, and highly motivated; it will be interesting to see whether text-based awareness can work in other distributed groups.

❖ Basic principles to generalize distributed awareness

Even if the specifics of these projects cannot be used widely, there are a few general principles that can have broader applicability in supporting distributed awareness. First is the importance of verbal communication, and the value of different forms and venues for discussion. For the most part, our findings reinforce previous results; however, it is worth noting the value of providing support for both 'formal' discussions (on the mailing list) and informal, ad-hoc talk on the chat system. It is also useful to know that written conversation can in some settings take the place of audio communication (a result that differs from other conclusions [30]. Second is the significance of overhearing as an awareness tool.
Although the usefulness of this behavior has been recognized in studies of audio channels, studies of textual communication have sometimes characterized these 'lurkers' negatively, as free riders.

In many circumstances, however, they may be simply acting as peripheral participants, gathering general awareness that helps them to keep in touch with the community and the project. Third is the value of broadcast communication. As seen in collocated situations, the ability to speak to an expected audience
rather than to a specific one had several advantages in finding the right people and allowing people to decide for themselves whether to respond. This principle and the one above suggest that designers should consider whether communication facilities should be public (like a chat server) rather than point-to-point (like instant messaging or private email).

❖ Open Source setting be better supportive platform

There were some indications of difficulties that were discussed in the interviews, even if these did not prevent people from maintaining awareness. For example, comments above mention the effort involved in reading the lists (particularly commit logs), the difficulty of managing one conversation in two communication channels (mailing list and chat), and the problems of looking for information in the mail archives. We are interested in whether developers' existing awareness support could be augmented without fundamentally changing it. We have several possibilities that we are currently discussing with developers:

• Mailing lists are time-consuming; we are looking at whether new representations of messages and threads can help to support group awareness with less effort.
• CVS commits are sometimes ignored due to time constraints; it may be possible to show dynamic awareness information from the CVS repository in a form that allows for easier browsing, filtering, and inspection.
• The splitting of communication between mail, chat, and issue tracker suggests potential for tools that link related conversational streams. This could allow conversations to be seen in the context of work artifacts without losing the public nature of the discussion.
• The idea of making things public could be extended to other types of interactions. Although this is already the basis for 'edit wear and read wear' approaches such as Augur [17], the idea could be extended to interactions with awareness information sources. For example, it could be valuable to visualize the frequency with which people look at different information in the CVS repository.
• Search tools could be designed with awareness queries in mind. Archives are valuable resources for group awareness, particularly for new developers, but it is not known how people look for awareness information in these kinds of databases. Mining the archives should be done with caution, however, given the likely reluctance to have certain types of social relationships made explicit.

FUTURE RESEARCH

Our future plans for Project Analyzer involve improvements and new directions in both the mining and the visualization components. The current version of the system primarily addresses those awareness issues that we saw in distributed projects, but the basic tools and approaches can be used for a variety of additional purposes. First, we currently visualize source code that is in the process of being edited, and therefore the source code may be inconsistent, incomplete and frequently updated. We are investigating techniques for improving the robustness and performance of the fact extraction process, and techniques for visualizing partial information given these circumstances. Our system also only records user edits to the method level. We plan to move towards even finer grained awareness so that we can handle concurrent edits in some situations. Second, the capturing and recording of developers' activities supports new software repository mining research in addition to supporting awareness. Developers normally change a local copy of the software under development, and periodically synchronize their changes with the shared software repository. Unfortunately, the developer's local interactions with the source code are not recorded in the shared software repository. With our finer-grained approach, the local interaction history of the developer is recorded and is available to be mined. Example software mining research directions include:
• Discovery of refactoring patterns. Analyzing local interaction histories may be useful for identifying novel refactoring patterns and coordinating refactoring that affects other team members.
• Discovery of browsing patterns. Local interaction history includes the developer's searching, browsing and file access activities. Analyzing this browsing interaction may be useful in supporting a developer in locating people or code exemplars.
• Discovery of expertise. Since the fact base contains facts from the Java API, we can determine what parts of that API each developer has used, and how often. It can now be possible to determine who has used a particular Java widget or structure frequently, and to build that knowledge into the development environment. We also plan to refine and expand the visualization component. Short-term work will involve testing the representations and filters to determine how the information can be best presented to real developers. Longer range plans involve extensions to the basic idea of integrating information about activities with information about project artifacts. For example, we plan to extend our artifact collection to include entities other than those in source code. Many other project artifacts exist, including communication logs, bug reports and task lists. We hope to establish additional facts to model these artifacts and to use the new artifacts and their relationships in the awareness visualizations. We can also extend our use of the interaction histories to other areas. As discussed above, recording developers' interaction history and extracting method call facts from the source code provides us with basic API usage information. We can present this information in the IDE to provide awareness of technology expertise.

Finally, we plan to extend the range of awareness information that can be seen in the visualizations. As mentioned above, displaying information about refactoring, browsing, and expertise may be useful to developers in a distributed project. Other possibilities include questions of proximity – "who is working near to me?" in terms of the structures and dependencies of the software system under development, and questions of scope and effect – "how many people will affect if I change this module?" Proximity is an important concept in software development because developers who near to one another (in code terms form) an implicit sub-team whose concerns are similar and whose interactions are more closely coupled [20]. Proximity groups are not defined in advance and change membership as developers move from task to task; therefore, it is often very difficult to determine who is currently in the group.

CONCLUSIONS

Open-source software development projects are examples of collaborative, distributed work where people are able to maintain awareness of each other and of others' activities. In this study we looked at requirements and mechanisms for group awareness on three open source projects. We found that distributed developers maintain both a general awareness of the entire team and more detailed knowledge of people with they are plan to work. The primary means for maintaining awareness were mailing lists and chat tools; we were struck by the capabilities of text-based communication for supporting awareness, and by the importance of the organizational culture in promoting the kinds of behavior that make good group awareness possible through these tools.

This study is one of the first to consider how awareness works in the real world. One thing that is clear from the study, in addition what we discuss above is that awareness is both complex and subtle. There are many leads in our data that we were unable to address here. These issues – such as the ways that non-English speaking developers use the lists, how occasional face-to-face gatherings assist group awareness, how reputation really affects mailing-list practices, what kinds of miscommunications arise in list-based discussion, or the ways that project size affect awareness mechanisms will be investigated as we look more closely at different parts of the data. We have presented a system to address some of the awareness problems experienced in distributed software development projects. Project Analyzer contains two main parts: a mining component and a visualization system. The system keeps track of fine-grained user activities through the use of a shadow repository, and records those actions in relation to the artifact-based dependencies extracted from source code. Second, visualizations represent this information for developers to see and interact with. The visualizations present a project overview, overlaid with visual information about people's activities. Although our prototypes have limitations in terms of project size, they can provide developers with much-needed information about who is working on the project, what they are doing and how the project is changing over time.

REFERENCES

[1] 1. Churchill, E., and Bly, S, It's all in the words: Supporting Work Activities with Lightweight Tools. Proc. ACM GROUP 1999, xx-yy.

[2] Fitzpatrick, G., Kaplan, S. Mansfield, T., Arnold D., and Segall, B., Supporting Public Availability and Accessibility with Elvin, JCSCW, 11(3), 447-474.

[3] Mockus, A., Fielding, R., and Herbsleb, J. Two Case Studies of Open Source Software Development: Apache and Mozilla, ACM ToSEM, 11, 3, 2002, 309-346.

[4] Whittaker, S., Terveen, L., Hill, W., and Cherny, L., The Dynamics of Mass Interaction, Proc. SCW 1998, 257-264.

[5] Herbsleb, J. and Grinter, R. Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software, Sept/Oct 1999, 63-70.

[6] Froehlich, J. and Dourish, P., Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. Proc. ICSE 2004, 2004, 387-396.

[7] Herbsleb, J., Mockus, A., Finholt, T., and Grinter, R., Distance, Dependencies, and Delay in a Global Collaboration, Proc. ACM CSCW 2000, 319-328.

[8] Raymond, E., The Cathedral and the Bazaar, O'Reilly, 2001.

[9] Ball, T., and Eick, S. Software visualization in the large. IEEE Computer, Vol 29, No 4, 1996.

[10] Chu-Caroll, M., and Sprenkle, S. Coven: Brewing better collaboration through software configuration management. Proc FSE-8, 2000.

[11] Cordy, J., Dean, T., Malton, A., and Schneider, K., Software Engineering by Source Transformation Experience with TXL, Proc. SCAM'01 - IEEE 1st International Workshop on Source Code Analysis and Manipulation, 168-178, 2001.

[12] Dix, A., Finlay, J., Abowd, G., and Beale, R., Human-Computer Interaction, Prentice Hall, 1993.

[13] Dourish, P., and Bellotti, V., Awareness and Coordination in Shared Workspaces, Proc. ACM CSCW 1992, 107-114.

[14] Elliott, M., and Scacchi, W., Free software developers as an occupational community resolving conflicts and fostering collaboration, Proc. ACM GROUP 2003, 21-30.

[15] Froehlich, J. and Dourish, P., Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. To appear, Proc. ICSE 2004.

[16] Gutwin, C. and Greenberg, S. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. Journal of Computer-Supported Cooperative Work, Issue 3-4, 2002, 411-446.

[17] Gutwin, C., Penner, R., and Schneider, K., Group Awareness in Distributed Software Development, to appear, Proceedings of ACM CSCW 2004, Chicago, 2004.

[18] Herbsleb, J., and Grinter, R., Architectures, coordination, and distance: Conway's law and beyond. IEEE Software, 1999.

[19] 19. Herbsleb, J., Grinter, R., and Perry, D., The geography of coordination: dealing with distance in R&D work. Proc. ACM SIGGROUP conference on supporting group work, 1999.

[20] Herbsleb, J., Mockus, A., Finholt, T., and Grinter, R., Distance, Dependencies, and Delay in a Global Collaboration, Proc. ACM CSCW 2000, 319-328.

[21] McDonald, D., and Ackerman, M., Just Talk to Me: A Field Study of Expertise Location Finding and Sustaining Relationships, Proc. ACM CSCW 1998, 315-324.

[22] Mockus, A., Fielding, R., and Herbsleb, J. Two Case Studies of Open Source Software Development: Apache and Mozilla, ACM ToSEM, 11, 3, 2002, 309-346.

[23] Monk, A., and Watts, L., Peripheral Participants in Mediated Communication, Proc. ACM CHI 1998, v.2, 285-286.

[24] Raymond, E., The Cathedral and the Bazaar, O'Reilly, 2001.

[25] Schummer, T., Lost and found in software space. Proc 34th HICSS, 2001.

[26] Schummer, T., and Schummer, J., TUKAN: A team environment for software implementation. Proc. OOPSLA 1999.

[27] Segal, L., Designing Team Workstations: The Choreography of Teamwork, in Local Applications of the Ecological Approach to Human-Machine Systems, P. Hancock, J. Flach, J. Caird and K. Vicente ed., Erlbaum, 1995, 392-415.

[28] Whittaker, S., Frohlich, D., and Daly-Jones, O., Informal Workplace Communication: What is It Like and How Might We Support It?, Proc. ACM CHI 1994, 131-137.

[29] B. Zimmermann and A. M. Selvin. A framework for assessing group memory approaches for software design projects. Proc. Conference on Designing interactive systems. 1997.

[30] Hill, W.C., Hollan, J.D., McCandless, J., and Wroblewski, D. Edit wear and read wear. Proc. ACM CHI 1992, 3-9.

[31] Kraut, R., and Streeter, L., Coordination in software development. CACM, 1995.

[32] Malton, A., Schneider, K., Cordy, J., Dean, T., Cousineau, D., and Reynolds, J., Processing Software

[33] Source Text in Automated Design Recovery and Transformation.

Proc. 9th International Workshop on Program Comprehension, 127-134, 20