Supporting Group Awareness in Distributed Software Development

Carl Gutwin, Kevin Schneider, David Paquette, and Reagan Penner

Department of Computer Science, University of Saskatchewan
Computer Science Department, University of Saskatchewan
57 Campus Drive, Saskatoon, SK
Canada, S7N 5A9
gutwin,kas,dnp972,rpenner @usask.ca

Abstract. 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 what tasks people have been working on. 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. ProjectWatcher observes fine-grained user edits and presents that information visually on a representation of a project's artifacts. The system displays general awareness information and also provides a resource for more detailed questions about others' activities.

1. Introduction

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 [2,10,11,14]. Even though projects are often organized to try and make modules independent of one another, dependencies cannot be totally removed [14]. 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 [10].

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 [10]. Current tools are focused around the artifacts of collaboration rather than people's activities (e.g., the 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.

We have built a system called ProjectWatcher that provides people with awareness information about others on the development team. The system is designed around our observations of the awareness requirements in several distributed software projects. We found that developers first maintain a general awareness of who is who and who is doing what on a project; and second, they actively look for information about people when they are going to work more closely with them. However, developers often have to use text-based sources to get that information.

ProjectWatcher observes and records fine-grained information about user edits and provides visualizations of who is active on a project, what artifacts they have been working on, and where in the project they have been working. This information about others' activities can help to improve coordination between developers and reduce some of the problems seen in distributed development.

In this paper, we introduce ProjectWatcher and describe its design and implementation. We first give an overview of the issues affecting collaboration in software development, and then discuss group awareness in more detail and the awareness requirements of a distributed development project. We then describe the two main parts of ProjectWatcher: a fact mining component that gathers developer activity information, and a visualization component that overlays activity data onto a representation of project artifacts.

2. Background

Although collaboration is an important research area of software engineering – where teams are common and where good communication and coordination are essential for success – little work has been done on group awareness in software development. Similarly, although 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. In the next sections, 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.

2.1 Collaboration Issues in Software Development

Collaboration support has always been a part of distributed development – 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 (e.g., CVS), require considerable time and effort (e.g., reading documentation), or depend on people's current availability (e.g., IRC).

Researchers in software engineering and CSCW 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 [14];
- communicate with others across timezones and work schedules [11];
- find partners for closer collaboration or assistance on particular issues [20];
- determine who has expertise or knowledge about the different parts of the project [24];
- benefit from the opportunistic and unplanned contact that occurs when developers are co-located, since there is little visibility of others' activities [10].

As Herbsleb and Grinter [10] state, lack of awareness – "the inability to share the same environment and to see what is happening at the other site" (p. 67) is one of the major factors in these problems.

2.2 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 [5]. Group awareness is useful for coordinating actions, managing coupling, discussing tasks, anticipating others' actions, and finding help [8]. 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* [22], in which watching another person work provides information as to their activities and plans; and *feedthrough* [4], 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 (e.g., [18,23]). 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 feedthrough 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 (e.g., TUKAN [21] or Augur [7]), but it is not clear that these tools really provide the awareness information that is needed by developers. As discussed in the next section, we based our tools and techniques on findings from a study of three distributed open-source projects.

3. Awareness Requirements in Distributed Development

Open-source software development projects are a good source of information about distributed development, since they are almost always collaborative and widely dispersed (in many cases, developers never meet face-to-face). To find out what the awareness requirements are for these long-running real-world projects, we interviewed several developers, read project communication, and looked at project artifacts from three open source projects [9]. We found that distributed developers do 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 that they plan to work with. However, developers maintain their awareness primarily through text-based communication – particularly mailing lists and chat systems.

The three open source projects we looked at are NetBSD (www.netbsd.org), Apache httpd (www.apache.org), and Subversion (www.tigris.org/subversion). We chose these projects because they are distributed, they are at least medium-sized in terms of both the code and the development team, and they all produce a product that is widely used, indicating that they have successfully managed to coordinate development.

An initial issue that we looked at was whether distributed projects can successfully isolate different software modules from one another such that awareness and coordination requirements become insignificant. There are two ways that dependencies can be reduced – by reducing the number of developers, or by partitioning the code. However, in the three projects we looked at, neither of these factors removed awareness requirements. There were at least fourteen core developers who contributed regularly to each project, and although there was general understanding that people work in ‘home’ areas, there were no official sanctions that prevented any developer from contributing to any part of the code. On Apache and Subversion in particular, development of a particular module was almost always spread across several developers.

The next issue studied was what types of awareness the developers maintained. We found two types: general awareness and more specific knowledge. First, developers maintain a broad awareness of who are the main people working on their project, and what their areas of expertise are. This information came from three sources: the project mailing list, where people can see who posts and what the topics of discussion are; the chat server, which provides similar information but in real time; and the CVS commits (sent out by email), which allowed developers to stay up-to-date both on changes to the project and the activities of different people. 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. We found that people use a variety of sources to gather this information, including project documentation, the records in the source code repository, bug tracking systems, and other people. Further details on this study can be found in [9].

Even though these open-source projects do successfully manage their coordination, our interviews also identified some problems with the way awareness is maintained. Two problems that we consider further in this paper involve watching CVS commits, and maintaining overall awareness about project members and their activities. Although the 'CVS-commit' mailing list provides the only information that is actually based on the project artifacts, several developers said that they do not follow them because they are too time-consuming to read. Developers also suggested that some of the information sources they use often go out of date, and that understanding the relationships between people and activities was often difficult. One developer stated that new members of the project in particular could benefit from tools that provided more information than what was currently available.

4. Project Watcher

We have developed an awareness system called ProjectWatcher to address some of the awareness issues that we have seen in distributed development projects. ProjectWatcher 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 plugin inside the Eclipse IDE. ProjectWatcher consists of two main parts – the mining component, and the awareness visualizations.

4.1 Mining Component

The mining component analyzes a project's source code to produce facts for use by the ProjectWatcher visualization displays. To gather developer activity information at a finer grain size than repository commits, a shadow CVS repository is maintained (see Figure 1). User edits are auto-committed to the shadow repository as developers edit source code files (e.g., on every save of the file). 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 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 repository 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.

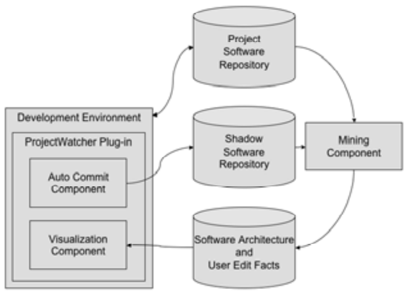

Fig. 1: User edit fact extraction.

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 (see 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 [15] program using syntactic pattern matching [3]. 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. Next, the method call facts are analyzed to determine which package and class the method that was called belongs to. 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 ProjectWatcher mining component (this is only done once for all projects).

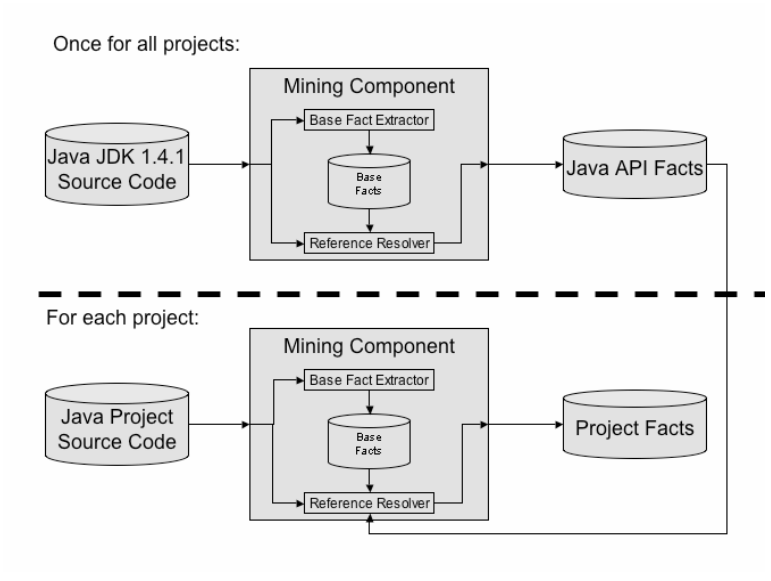

Fig. 2: Software architecture fact extraction from Java projects

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 factbase. 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 factbase.

The complete factbase 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 factbase storage depends on the size of the code; for example, the Java Development Kit 1.4.1 contains 202 package facts, 5,530 class facts, 47,962 method facts, and 106,926 method call facts.

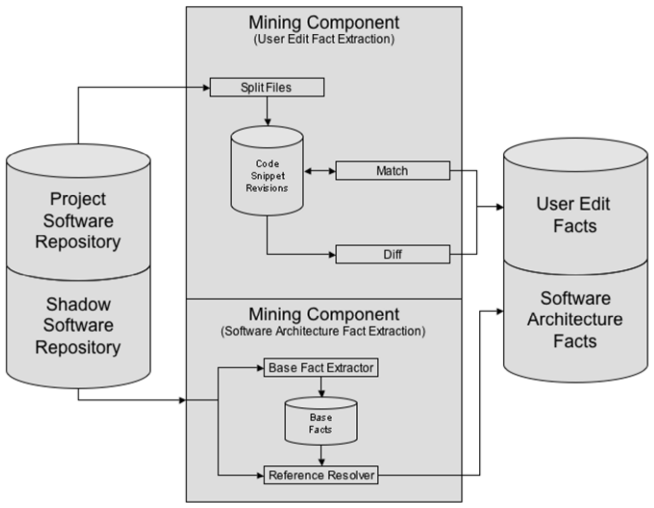
Mining Component
(User Edit Fact Extraction)
Split Files
Code Snippet Revisions
Match
Diff
Project Software Repository
Shadow Software Repository
User Edit Facts
Software Architecture Facts
Mining Component
(Software Architecture Fact Extraction)
Base Fact Extractor
Base Facts
Reference Resolver

Fig. 3: User edit fact extraction.

4.2 Visualization of activity and commits

ProjectWatcher's activity awareness display visualizes team members' past and current activities on project artifacts (see Figures 4 and 5). The goals of this display are:

- to give collaborators an overview of who works on the project
- to provide a general sense of who works in what areas
- to allow changes (i.e., commits) to be tracked without much effort
- to provide more detail when the user wants to look more closely.

The display uses the ideas of edit wear, interaction histories, and overviews. *Edit wear* is a concept introduced by Hill and colleagues [13]. Their overall motivation is the question of how computation can be used to improve "the reflective conversation with work materials" (p. 3), 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. Starting with this idea of 'object wear,' their research proposes an 'informational physics' in which the visual appearance of an object arises not from everyday physical laws, but from informational rules that are semantically useful. Their notion of physics has objects explicitly show different aspects of their use over time – that is, their interaction history:

> The basic idea is to maintain and exploit object-centered interaction histories: record on computational objects…the events that comprise their use…and display useful graphical abstractions of the accrued histories as part of the objects themselves." ([13], p. 3)

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 ProjectWatcher, 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 (e.g., by colour), 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.

ProjectWatcher uses the extracted fact base to create a visual model of what each developer is doing in the project space. Project artifacts are shown in a simple stacked fashion that displays packages, files, classes, and methods. We chose this method of organization because it is much more compact than other approaches, such as class diagrams or dependency graphs. With the stacked representation, even a small overview can completely display projects with up to several hundred files (e.g., Figure 4 shows 322 files); in larger projects, developers can collapse particular packages to save space. The drawback with the stack is that there is little contextual information available to help users determine which artifact is which. To try and reduce this problem, artifacts are always stacked by creation date, so that their location in the overview is fixed, and can over time be learned by the user. We are also experimenting with allowing users to reorganize the display, so that they can arrange and group the artifacts in ways that are more meaningful to them.

On this basic overview representation, we overlay awareness and change information. First, each developer is assigned a unique colour, and this colour can be added to the blocks in the overview based on a set of filters. Common filters that involve developer information include who has modified artifacts most recently, and who has modified them most often. Other filters exist as well, such as one that shows time since last change (see Figure 5). Second, we show a summary of the activity history for each artifact with a small bar graph drawn inside the object's rectangle; bars represent amount of change to the class since its creation. More information about an artifact can be obtained by holding the cursor over a rectangle: for example, the name of the class and a more detailed bar graph.

Change information can be shown in addition to information about developers. The system highlight artifacts (using coloured borders) if they have changed recently – this provides users with dynamic information about commits to the project. When a change occurs to the CVS repository, the changed files are highlighted in the overview representation. More details about the change can be seen using the popup detail window, and further information (such as the difference between the two versions) can be seen through a context menu.

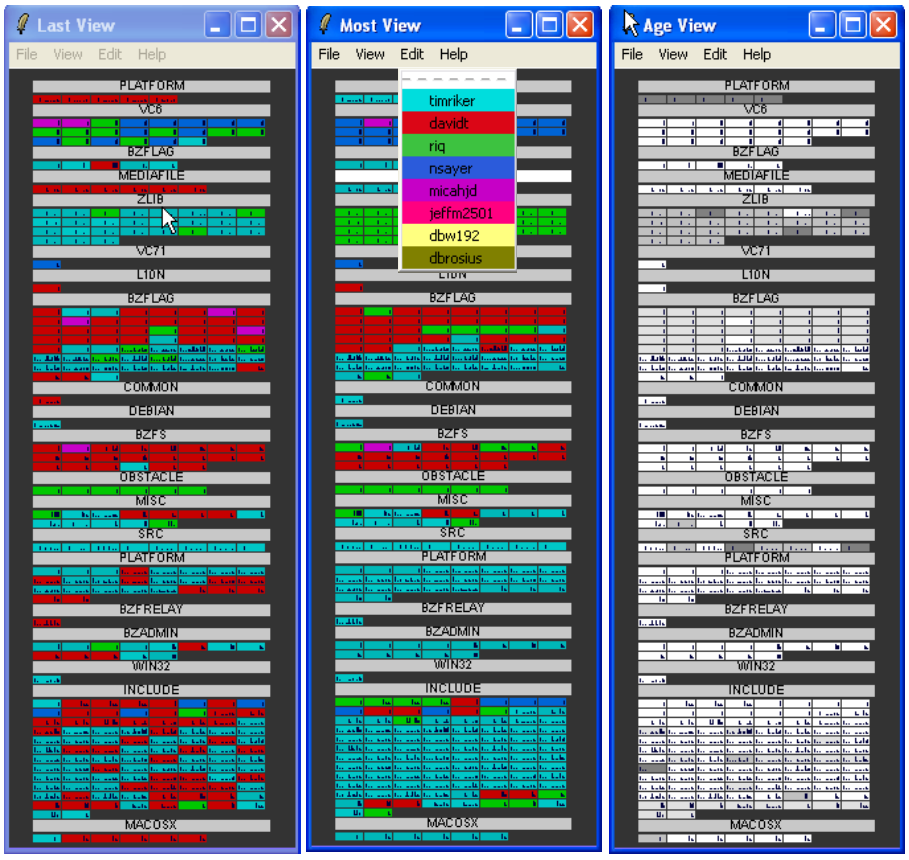

Fig. 4. Project overviews showing directories (grey bars) and files (coloured blocks) for a medium-sized game project with 322 files. Three types of filters are shown: at left, block colour indicates who changed the file most recently; at middle, colour shows who has changed the file most often; at right, grey level indicates the amount of time since last change. In each block, the bar graph shows the edit history since the start of the project. Developer colours are shown in a menu. Note that normally only one window would be used, with the filter changed through a menu selection.

The overview displays help developers to answer a variety of questions about the project and about the activities of their collaborators. For example, it can be seen that the developers timriker (light blue) and davidt (red) are currently active (since they have each been the last to touch several files), and are core developers on the project (since they are both the most frequent committer for many files). We can also see that developers riq (green) and nsayer (dark blue) are each likely responsible for one main module in the project, since they are the most frequent for all the files in a particular directory. Two other people, dbw192 (yellow) and dbrosius (brown) are neither recent or frequent committers, since neither filter shows any files in their colour. Finally, we can see from the ‘age’ filter (Figure 4, right) that most of the project has recently been changed, since most of the blocks are white or light grey.

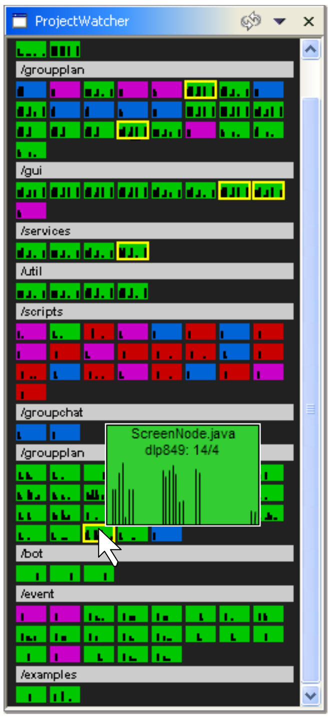

Fig. 5. ProjectWatcher as an Eclipse IDE plugin (www.eclipse.org), showing highlights (yellow borders on blocks) to indicate others’ recent changes, and popup window to show more detail about a particular file.

The highlights (see Figure 5) provide an analogue to the CVS-commits mailing list, but with considerably less effort. As can be seen in the figure, there are six files that have been changed since the local user last updated files from the repository. It is easy to determine how much change is occurring, and in general where it is happening. By holding the mouse cursor over any of these blocks, the developer, can get more information about what file has been changed, who committed the most recent change, and the number of lines added and deleted in the change (the ‘14/4’ in the popup indicates that 14 lines were added, and 4 deleted).

5. Comparison to 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 (e.g., [2,6,19]). However, only a few systems combine information about people’s activities with representations of the project artifacts. Two that do this are Augur [7] and TUKAN [20,21].

TUKAN is one of the first systems to explicitly address the question of awareness in software development. The basic representation used in TUKAN is 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 ProjectWatcher is in the use of an overview; where TUKAN presents relevant information about others who may be encroaching on a developer’s current location, ProjectWatcher provides a general overview of the entire project.

Augur is a system similar to Ball and Eick’s SeeSoft [1], 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. ProjectWatcher 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 ProjectWatcher’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, ProjectWatcher 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 ProjectWatcher as more suited to day-to-day activities on a collaborative project, and Augur to specific investigations where developers wish to explore the history of the project in more detail.

6. Future Research

Our future plans for ProjectWatcher 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*. Analysing local interaction histories may be useful for identifying novel refactoring patterns and coordinating refactorings that affect other team members.
- *Discovery of browsing patterns*. Local interaction history includes the developer's searching, browsing and file access activities. Analysing this browsing interaction may be useful in supporting a developer in locating people or code exemplars.
- *Discovery of expertise*. Since the factbase 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 I 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. We will address this problem by extending the ProjectWatcher visualizations to make it easier to see proximity-based groups.

6. Conclusions

We have presented a system to address some of the awareness problems experienced in distributed software development projects. ProjectWatcher 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.

Acknowledgements

The authors would like to thank IBM Corporation for supporting this research.

References

1. Ball, T., and Eick, S. Software visualization in the large. *IEEE Computer*, Vol 29, No 4, 1996.
2. Chu-Caroll, M., and Sprenkle, S. Coven: Brewing better collaboration through software configuration management. *Proc FSE-8*, 2000.
3. 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.
4. Dix, A., Finlay, J., Abowd, G., and Beale, R., *Human-Computer Interaction*, Prentice Hall, 1993.
5. Dourish, P., and Bellotti, V., Awareness and Coordination in Shared Workspaces, *Proc. ACM CSCW 1992*, 107-114.
6. Elliott, M., and Scacchi, W., Free software developers as an occupational community: resolving conflicts and fostering collaboration, *Proc. ACM GROUP 2003*, 21-30.
7. Froehlich, J. and Dourish, P., Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. To appear, *Proc. ICSE 2004.*
8. 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.
9. Gutwin, C., Penner, R., and Schneider, K., Group Awareness in Distributed Software Development, to appear, *Proceedings of ACM CSCW 2004*, Chicago, 2004.
10. Herbsleb, J., and Grinter, R., Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 1999.
11. 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.

12. Herbsleb, J., Mockus, A., Finholt, T., and Grinter, R., Distance, Dependencies, and Delay in a Global Collaboration, *Proc. ACM CSCW 2000*, 319-328.
13. Hill, W.C., Hollan, J.D., McCandless, J., and Wroblewski, D. Edit wear and read wear. *Proc. ACM CHI 1992*, 3-9.
14. Kraut, R., and Streeter, L., Coordination in software development. *CACM*, 1995.
15. Malton, A., Schneider, K., Cordy, J., Dean, T., Cousineau, D., and Reynolds, J., Processing Software Source Text in Automated Design Recovery and Transformation. *Proc. 9th International Workshop on Program Comprehension*, 127-134, 2001.
16. 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.
17. 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.
18. Monk, A., and Watts, L., Peripheral Participants in Mediated Communication, Proc. ACM CHI 1998, v.2, 285-286.
19. Raymond, E., The Cathedral and the Bazaar, O'Reilly, 2001.
20. Schummer, T., Lost and found in software space. *Proc 34th HICSS*, 2001.
21. Schummer, T., and Schummer, J., TUKAN: A team environment for software implementation. *Proc. OOPSLA 1999.*
22. 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.
23. 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.
24. B. Zimmermann and A. M. Selvin. A framework for assessing group memory approaches for software design projects. *Proc. Conference on Designing interactive systems*. 1997.

Discussion

[Bonnie E. John] You chose no to look at video or IM Buddy lists, is that because prior research suggests that that is not where the action is, or was it easier not to do that, or what?

> [Kevin Schneider] We were interested in the software artefacts and what we could get from that! Other people in the CSCW field are working on other aspects such as the ones you mention. The field does not really know where the bang for the buck is.

[Bonnie John] You mentioned scalability! How big does it scale and do you have ideas of how you could chunk or aggregate to allow you to scale further? Are we talking about 10 person projects with 10,000 lines of code or a 100 person project with 1,000,000 lines of code?

> [Kevin Schneider] It is a big issue! I think the visualisation might not scale and that is why we are trying to think of other metaphors! Currently 10,000 to 100,000 would probably be the limit! Currently we use relatively little screen space and the projects we have looked at does not seem to need more than that! Other studies have shown that even large projects such as Linux tends to be organised around specific parts of the code and that might help solve the scalability problem you mention! Maybe it is software architecture that will have to solve that problem!

[Peter Forbrig] I like your tool very much. What about the software developers? Did they like to be tracked in this way?

> [Kevin Schneider] Because we were looking at open source projects there was no problem with privacy. Their community is willing to publish all activities. We can combine our approach with techniques to achieve privacy, but we did not look at it up to now.

[Bonnie John] Are real people using it and would they hate you if you took it away from them?

> [Kevin Schneider] Only internal people are using it, and we do not know if they would hate us if we took it away!