Proceedings of:

Supporting the Social Side of Large Scale Software Development

a
CSCW Workshop
Banff, AB
November 2006

Organizers:
Li-Te Cheng, IBM Research
Anthony Cox, Dalhousie University
Rob DeLine, Microsoft Research
Cleidson de Souza, Universidade Federal do Para
Kevin Schneider, University of Saskatchewan
Janice Singer, National Research Council of Canada
Margaret-Anne Storey, University of Victoria
Gina Venolia, Microsoft Research

Table of Contents[1]

[1] Papers are listed in the order in which they were received.

Attendees:

Name	Organization	Email
Ban Al-Ani	UC Irvine	balani@uci.edu
Jorge Aranda	University of Toronto	jaranda@cs.toronto.edu
Flore Barcellini	INRIA Eiffel Team	flore.barcellini@inria.fr
Andrew Begel	Microsoft Research	abegel@microsoft.com
Marcelo Cataldo	Carnegie Mellon University	mcataldo@andrew.cmu.edu
Li-Te Cheng	IBM Research	li-te_cheng@us.ibm.com
Davor Cubranic		cubranic@acm.org
Daniela Damian	University of Victoria	danielad@cs.uvic.ca
Rob DeLine	Microsoft Research	rob.deline@microsoft.com
Cleidson de Souza	Universidade Federal do Para	cdesouza@ufpa.br
Françoise Détienne	INRIA	francoise.detienne@inria.fr
Steve Easterbrook	University of Toronto	sme@cs.toronto.edu
Robert Elves	University of British Columbia	relves@cs.ubc.ca
Shannon Goodman	Smart Technologies	shannong@smarttech.com
Wolfgang Graether	Fraunhofer FIT	wolfgang.graether@fit.fraunhofer.de
Jim Herbsleb	Carnegie Mellon University	jdh@cs.cmu.edu
Bryan Kirschner	Microsoft Research	bryankir @microsoft.com
Andrew Ko	Carnegie Mellon University	ajko@cs.cmu.edu
Birgit Krogstie	Norwegian University of Science and Technology	birgitkr@idi.ntnu.no
Kumiyo Nakakoji	RCAST, Univ. of Tokyo	kumiyo@kid.rcast.utokyo.ac.jp
Masao Ohira	NAIST	masao@is.naist.jp
David Redmiles	UC Irvine	redmiles@ics.uci.edu
Kevin Schneider	University of Saskatchewan	kas@cs.usask.ca
Sadat Shami	Cornell University	ns293@cornell.edu
Anders Sigfridsson	University of Limerick	anders.sigfridsson@ul.ie
Jonathan Sillto	University of Calgary	sillto@cpsc.ucalgary.ca
Janice Singer	National Research Council Canada	janice.singer@nrc cnrc.gc.ca
Suzanne Soroczak	University of Washington	suzka@u.washington.edu
Margaret- Anne Storey	University of Victoria	mastorey@uvic.ca
Andrew Sutherland	University of Saskatchewan	als153@mail.usask.ca
Gina Venolia	Microsoft Research	gina.venolia@microsoft.com
Yasuhiro Yamamoto	University of Tokyo	yxy@kid.rcast.u-tokyo.ac.jp

The Role of Science in Supporting Software Development

Andrew J. Ko
5000 Forbes Avenue, Pittsburgh PA, 15213
Human-Computer Interaction Institute
Carnegie Mellon University
ajko@cs.cmu.edu

ABSTRACT

Discusses the importance of scientific explanations in tool design, and various ways of forming such explanations.

Categories and Subject Descriptors

H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms

Design, Human Factors, Experimentation.

Keywords

Empiricism, science, design, tools, evaluation, notation, theory, measurement, prototyping, experts, ethnography, collaboration.

1. INTRODUCTION

The primary focus of this workshop is to reflect on how tools can support the social side of software development. In service of this goal, rather than using this space to espouse my own ideas about how this might be done, I would instead like to reflect on the methods by which we invent such tools.

It is difficult to invent useful tools without some understanding of how people develop software. Even the most biased of tool designers have some model in their minds of what is important to software developers. Of course, these models are largely based on personal experience. While experience can be a valuable form of inspiration, what differentiates research from experience is science—and scientists seek to *explain*.

Therefore, while *descriptions* of the social side of software development have captured many of its modern practices, descriptions are insufficient for design. We need to know *why* software development is social. Is it because developers prefer to be social or because they need to be? What do developers gain by communicating with their peers? We know that some of this is to maintain awareness [2] and some is to learn from experts [3]—but awareness and knowledge of what? Are coworkers the only source for such information, or just the preferred source?

These questions are more than scholarly: the explanations we derive by investigating these questions are fodder for design. The more we understand *why* developers are social, the better that tools can match developers' needs. The better we can explain *why* developers seek awareness and expert knowledge, the better we can evaluate tools and articulate their tradeoffs.

But how can we explain these phenomena? Empiricism and observation are essential tools, but I would argue insufficient. One of their limitations is that the forms of explanations that they generate—models, theories, diagrams, etc.—rarely do justice to reality. We need to proceed one step further

3. EXPLAINING THROUGH DESIGN

Prototyping new technologies has played an equally important role in my studies. As with any design, my inventions did not follow directly from the understanding I have gained through observation. Rather, they are a culmination of the understanding I have gained about software development, both from my own investigation and from the decades of research that came before.

Consider the Whyline [7], the first tool that I worked on in my doctoral work. The idea behind this debugging tool was to help developers ask questions about their program's output and reveal their implicit assumptions about what had occurred at runtime. While I used my observation of the Building Virtual Worlds class discussed earlier for inspiration, the idea ultimately originated from several months of reflection and reasoning about the work that I observed. and a careful study of other debugging tools described in the literature The understanding and theories I had gained from observations helped me to *evaluate* and *test* the merits of my ideas, but not to *form* them.

Furthermore, because the theories behind the tool's design were incomplete, people used the Whyline in surprising ways. For example, one of the participants in my evaluation study had used the Whyline a few times and it had pointed out some of the assumptions she had made about what happened while her program was executing. The next time she began to ask the tool a question, she hovered over the "Why" button, but said, "I don't even need to ask. I think I made the same assumption that I did last time." The tool was introducing participants to the very same notion of assumptions that had inspired the Whyline's design—in this sense it *embodied*, *validated* and even *elaborated* the theories that motivated it.

Another tool I was involved in designing, Jasper [1], followed a similar trajectory. The original idea was inspired by a finding that developers gathered many little pieces of a program for a particular task, but had no way to gather them together in a single place [5]. This led to navigational overhead, as they navigated back and forth between code snippets that were distributed amongst several files. While my colleague designed and implemented the tool, I was busy at Microsoft, watching developers do work. As I watched them consult each other for knowledge about what code was relevant to a bug report or feature, I realized that being able to gather together snippets was not only helpful in reducing navigational overhead, but a fundamentally important way to share the *context* of one's task with coworkers. This new understanding changed the purpose of the tool in my mind: rather than just a navigational aid, it was a medium for externalizing and sharing task context. Had I noted invented the idea, this realization would not have been possible.

4. EXPLAINING THROUGH EVALUATION

Understanding and invention are vital ingredients in improving software engineering, but they are little without a notion of success to guide our research efforts. Is my tool helpful? Is it effective? Does it improve productivity? Will people adopt it? These are the criteria by which we separate successful and unsuccessful design. Unfortunately, unlike success measures in other engineering disciplines, these are difficult to measure and not necessarily the same as those which users of our tools employ to evaluate tools.

One view on this issue is that "good" and "productive" should be defined by what a *developer* thinks is good and productive. Who better to evaluate the utility and fit of a tool than the people most familiar with a job's complexities? The challenge of this approach is that as researchers, we must often settle for creating prototypes rather than fully functional and usable products. This makes it difficult to know whether problems observed in evaluations are due to the tool's incompleteness or some underlying inadequacy.

Of course, a measure based on developers' reactions also suffers from bias, subjectivity, and considerable variation. There may be absolute measures of success that avoid these problems. For example, to what degree did a team create what it *intended* to create? Did the rates of information acquisition and decision making increase? Did the right quality attributes improve with the intervention? Although such measures are extremely difficult to compute, they may be necessary to pursue if we wish to clearly articulate the merits of our ideas to ourselves and to the world.

Whatever the merits of our measurements or the results of our evaluations, the key result of these studies is the elaboration of our explanations. By completing this loop between design and understanding, we inevitably improve the designs in our minds.

5. CONCLUSIONS

To support the social side of software development—or more appropriately, to decide whether to do so and why—researchers must explain why developers rely on each other in the ways that they do. As we rise to this challenge, let us remember that the *diversity* of our ideas, methods, skills and experiences are our greatest strength.

6. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under NSF grant IIS-0329090 and as part of the EUSES consortium under NSF grant ITR CCR-0324770. The first author was supported by an NDSEG fellowship.

7. REFERENCES

[1] Coblenz, M., Ko, A. J., Myers, B. A. (2006). Carnegie Mellon University, CMU-HCII-06-107.

[2] Gutwin, C., Penner, R. and Schneider, K. (2004). Group Awareness in Distributed Software Development. CSCW, Chicago, IL, 72-81.

[3] Hertzum, M. (2002). The Importance of Trust in Software Engineers' Assessment of Choice of Information Sources. Information and Organization, 12(1), 1-18.

[4] Ko, A. J. (2003). A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment. CHI, Fort Lauderdale, FL, 1036-1037.

[5] Ko. A. J., Myers, B.A., Coblenz, M. and Aung, H. H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. Transactions on Software Engineering, to appear.

[6] Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. VL/HCC, Rome, Italy, 199-206.

[7] Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. CHI, Vienna, Austria, 151-158.

[8] Sevenster, A. (1992). Collected Works of A.M. Turing: Mechanical Intelligence, Volume 1. Elsevier, New York:NY

Connecting People in Social Networks using Requirement Explorer

Irwin Kwan Daniela Damian
University of Victoria
3800 Finnery Road
Victoria, British Columbia, Canada
{irwink,danielad}@cs.uvic.ca

ABSTRACT

To help support communication among co-located and distributed teams, we present *Requirement Explorer*, which supports the collaboration of contributors working on a set of inter-dependent requirements. The Requirement Explorer uses social networks to display information about who a contributor should be communicating with when developing a requirement. The Requirement Explorer also displays the quantity of communication that has actually happened regarding a contributors requirements. Currently, we acquire task data from Bugzilla repositories, but we plan to develop the tool to use more forms of communication, such as E-mail, project documents, and source code to build more complete and more accurate social networks. Applications of this tool include making a contributor aware of who he should be communicating with when working on his requirements, making a contributor aware of who has contributed to the development of his requirements, and identifying gaps in communication between contributors that should be coordinating with each other.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*tools*

General Terms

Management, Human Factors

Keywords

Communication, collaboration, awareness, collaborative tools, large-scale distributed software development, visualization, social network analysis, change management

1. INTRODUCTION

No matter what process a software project follows, a large amount of collaboration must occur among the project team members, or contributors, to ensure that the artifacts are built properly. However, collaboration in software development can be difficult because of constant changes in requirements as well as project artifacts such as source code. When changes occur, every contributor working on the artifact and related artifacts should be notified in a timely manner so that each one can respond accordingly. When a contributor is constantly notified in a timely manner about change that affect him, he is said to be aware. Because a software contributor needs to coordinate with many contributors about various requirements, awareness can be very difficult to maintain. For instance, a manager, when scheduling a meeting about a requirement, may forget to call a contributor who has contributed heavily to testing it. Maintaining an up-to-date awareness with respect to requirements changes and resulting impact is particularly difficult in geographically distributed development environments where collaborative development activities do not benefit from the low-cost interactions specific to traditional co-located settings [5, 3].

To promote better awareness, we present *Requirement Explorer*, a prototype of an awareness system that supports more effective coordination in co-located or distributed software development. The Requirement Explorer tracks a persons current requirements and displays a list of people that he should be contacting regarding each one.

Drawing on our own prior research results [6, 7] as well as theories of coordination in software development [1], our approach to the design of this awareness system consists of developing and leveraging a social network that identifies the stakeholders who should be communicating with each other regarding a particular requirement. The requirements-based awareness system monitors the development environment and dynamically builds two types of social networks: 1) a person-based social network, which we call the *ideal-coordination social network* (ICSN), that indicates those who *should be communicating* because they are working on the same or inter-dependent requirements, and 2) a social network indicating those who are *actually communicating* with each other regarding the same or inter-dependent requirements, which we call the *actual-coordination social network* (ACSN). The ACSN is used for 1) providing up-to-date information on who is working on the same or inter-related requirements, particularly useful for expertise seeking and automated propagation of change information to the related project members, and 2) identifying gaps in the ACSN versus the ICSN so that corrective actions for more effective team coordination can be taken during the development of

Workshop on Supporting the Social Side of Large-Scale Software Development 2006 Banff, Alberta Canada

a requirement rather than later in the project.

The paper is organized as follows. We discuss further motivation for the tool and related work in Section 2. We then present the tool, *Requirement Explorer*, in Section 3. We discuss challenges and future work in Section 4, and conclude the paper in Section 5.

2. MOTIVATION AND BACKGROUND RESEARCH

2.1 Maintaining Awareness in Teams

It has been shown that a lack of awareness can cause breakdowns in communication, which consequently lead to rework in software artifacts [2, 5, 6]. In situations where there is a lack of awareness, communication is unnecessarily repeated, and changes require more time to coordinate [5]. As development proceeds over time, more contributors are involved in the development of a requirement than initially planned [6, 1, 4]. A contributor external to the team may contribute valuable expertise to development and therefore become involved with that requirement, but the project plan may neglect to include every member of the team who ends up working on that requirement. In this paper, we refer to contributors who are added to development after a plan has been laid out adds. The fact that there are changes in the number of contributors who work on a requirement can cause problems when trying to maintain awareness. A notification system or a person in charge of sending notifications may neglect to include an add, therefore delaying that persons awareness of the project.

2.2 Social Network Analysis in Software Engineering

The use of social network analysis is gaining recognition as a method to study software engineering organizations. Izquierdo uses social network analysis to study the participation of contributors within a particular feature that was to be implemented in the software system [6]. The case study tracked communication among team members regarding the feature and found that over time, a large number of contributors who were not assigned to work on the feature were participating in the development of the feature. Erhlich, et al. [4] studied three independent development teams and used social network analysis to analyze how team members in these teams communicated with each other, and why. Using social networks based on communication among contributors, the authors discovered a number of factors that influenced strong communication ties. They found that that accessibility, which means that a contributor is near by and easily contacted, as well as peripheral awareness, which means that a contributor is aware of this persons skill set, were the strongest influencing factors on communication. These two case studies reveal that social network analysis can reveal interesting coordination patterns within software engineering organizations.

3. REQUIREMENT EXPLORER: A TOOL SUPPORTING COLLABORATION

Requirement Explorer is a tool that displays social networks of contributors in an organization who work on interdependent requirements. This tool provides a visual representation of coordination congruence first presented by

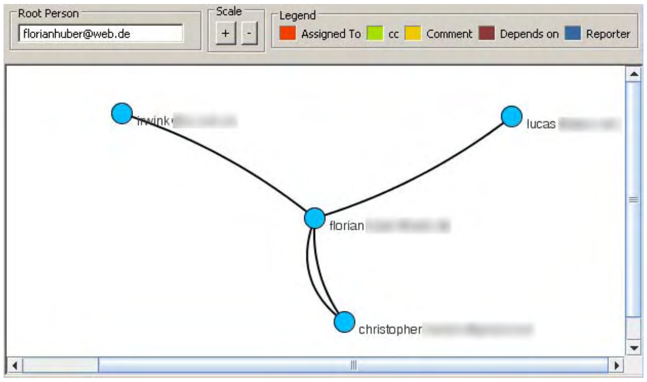

Root Person: florianhuber@web.de

Figure 1: An ideal-coordination social network diagram from Requirement Explorer. florian, the user, should be collaborating with irwink, lucas, and christopher. The two connections between florian and C*christopher* indicates that they have two sets of inter-related requirements to discuss.

Cataldo [1]. In this respect, our development is theory-driven. We incorporate two theories into this tool to show their viability. First, we draw an ICSN that displays the requirements that a person is currently working on. This network is useful for not only a contributor in an organization working on many requirements, but also for a manager who wishes to view who should be communicating with whom. Second, we use the idea of congruence presented by Cataldo to highlight differences between the ICSN and the ACSN.

Although the concept of congruence in Requirement Explorer and in Cataldos work is similar, our intention differs significantly from Cataldos. The analysis that Cataldo does with congruence is an attempt to identify the factors of communication on the cost of a software project. In contrast, we use congruence and present it visually to a contributor so that the contributor is more aware of whom he should be communicating with. We display, visually, where we may have potential gaps in communication that must be filled. By looking at a visual representation of a social network, we can easily see who needs to communicate with whom, and identify where breakdowns in communication are occurring. Using visualization techniques, we make identifying gaps in communication accessible to every contributor in a project. We also use the diagram to identify where we may have adds in a requirements team, and compensate by ensuring this person is involved appropriately in coordination and synchronization.

3.1 Requirement Explorer Implementation

The current implementation of Requirement Explorer uses Bugzilla as its source of data. Although we draw on bug data, Bugzilla does not actually require that a Bugzilla bug be a bug, and in many open-source projects, Bugzilla bugs are actually feature requests or requirements. In the future, we plan to expand the system to work with requirements databases. In the meantime, we have used the Eclipse Bugzilla repository as a source for our data.

To assign the ICSN, we use the assigned-to field on the We identify interdependencies among requirements using the depends-on field in a requirement. In a project setting, we would use information from identified dependencies among

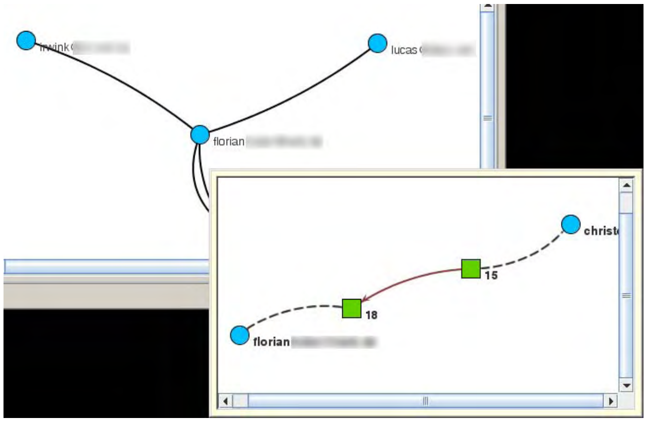


Figure 2: A popup from the ideal-coordination network showing the relationship among the requirements that link the two contributors. The solid line is a dependency arrow. The dashed line indicates that the contributor is assigned to the bug. The popup appears when you hover the mouse cursor over an edge on the diagram.

requirements in a project plan.

By default, we load up the ICSN for a persons list of assigned requirements, shown in Figure 1. This network displays the user in the centre, and shows the connection to each person that the contributor should maintain contact with when working on his requirements. The ICSN connects a person to a requirement using the Assigned To relationship in, and connects a requirement to another requirement using the Depends On relationship. If the centre contributor and another contributor are connected because of more than one requirement, then one link for each unique set of dependencies is displayed. To acquire more information about the requirement, the user can hover over the communication link to see a popup graph that displays the bugs that connect the contributor with the other person, as seen in Figure 2. The pop-up graph displays the dependency path of the bugs that connect the two contributors.

The actual-coordination network displays the user as the central contributor in addition to every person that he has communicated with about the requirement, similar to the social network in [7]. We identify connections by analyzing the Reported relationship, which identifies who first posted the requirement, the Commented On relationship, which is created when a person posts a comment on a requirement, and the Carbon Copy (CC) relationship, which means that a person receives a copy of every comment posted about this requirement. We see an example of a communication match between florian and christopher in the popup displayed in Figure 3 because they were identified as being connected in the ICSN. Note that in the main diagram, sminto is a new person in the network; we identify sminto as an add to florians requirements-related contacts.

Figure 4 shows another popup graph from the actual-coordination social network, highlighting a communication mismatch between irwink and florian about Bug 19. In this situation, irwink reported a requirement, and florian is receiving comments on this requirement, which is an interaction that was not predicted from the ICSN. The presence of these dynamic interactions may indicate that irwink knows something that florian does not, or could suggest that one of the contributors is seeking expertise from another.

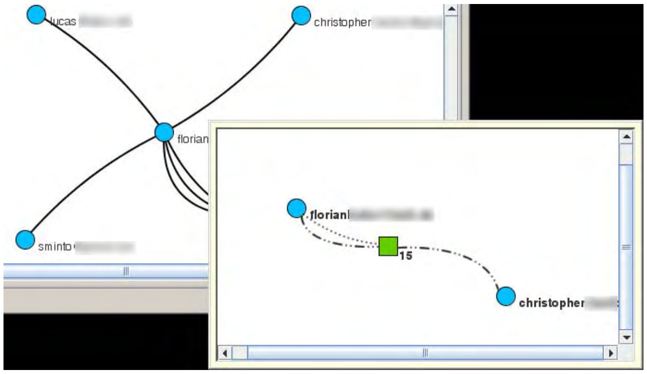


Figure 3: An actual-coordination social network diagram from Requirement Explorer with a popup. In the popup, we see that florian, the user, should be collaborating with irwink, lucas, and christopher. The two edges between florian and christopher indicates that they have two sets of inter-related requirements to discuss.

By providing visual feedback, we make analyzing coordination patterns easy for every member of the development team. Although not shown in the diagrams, we intend to implement visual notifications to highlight communication matches, communication mismatches, and adds. We can easily identify where communication breakdown may be occurring, and we can also help contributors maintain communication among those who have contributed to the development of a requirement.

4. CHALLENGES AND FUTURE DEVELOPMENT

The development of Requirement Explorer is in its infancy, but a number of features draw from coordination theory and awareness notification theory.

We intend to support better social network overlapping features in the visualization to better highlight the differences between the ideal-coordination social network and the actual-coordination social network. For instance, we plan to use glowing edges to highlight where the coordination requirements match up, and red edges to identify where communication may be missing.

Another feature we wish to incorporate is to save changes over time to both social networks, and view a timeline of changes in the networks. We can view the changes to the ideal-coordination social network through additional assignments, or through requirement interdependencies. Viewing changes in the actual-coordination social network reveals how people communicate over time, and can also highlight adds to the requirement team.

By using more sources of data, we can further improve the quality of both the ideal-coordination social network and the actual-coordination social network. The ideal-coordination social network information can come not only from a problem-reporting system, but also from a project-management tool. The actual-coordination social network can be generated from additional sources of information, including E-mail mes-

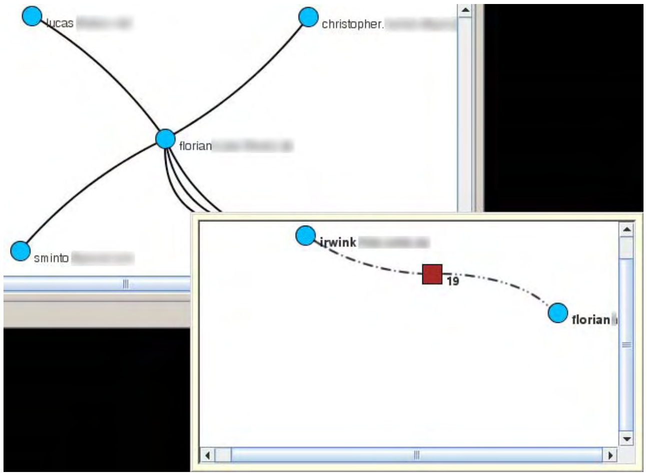


Figure 4: An actual-coordination social network diagram from Requirement Explorer with a popup. The popup shows an example of a communication mismatch, where communication occurred between irwink and florian despite the fact that this was not predicted in the ideal network.

sages and source code check-ins. A significant challenge will be how we can relate these sources of communication and artifacts to a requirement. We may explore keyword analysis or information retrieval techniques as

Finally, by using the data gathered by the tool, we can employ automated mechanisms for awareness notification. The Requirement Explorers usefulness does not need to stop at the visualization level. For example, we may want to send customized notifications depending on what a contributor has contributed to the requirement. We may include helper tools for a contributor who wishes to announce a change about the requirement.

With feedback from the community, we hope to improve the existing functionality of Requirement Explorer and add additional, useful features to fulfill its purpose of informing each contributor about the people they should be communicating with.

5. CONCLUSIONS

The Requirement Explorer is a tool used to promote awareness among contributors working on the same requirement. The Requirement Explorer features two primary characteristics. One, it displays an ideal-coordination social network to display who should be coordinating with whom in an ideal situation. Two, it displays an actual-coordination social network, similar to the requirements-centred social network [7], to identify who has actually communicated with whom. By comparing the two networks, a user can easily identify gaps in communication. Using these networks, a user is also aware of whom he should maintain contact with when working on a particular requirement.

Although the prototype uses data from the Bugzilla problem reporting system, we plan to extend it to include additional sources of information, such as source code repositories, requirements documents, and project plans. We believe that this tool helps promote awareness among contributors in a project. A contributor, using the ideal-coordination social network can easily identify who he needs to notify when working on a requirement, and can send the appropriate message to those who are dependent on a requirement he is working on. Once a contributor has sent these notifications, he can monitor the actual-coordination social network in order to ensure that each member of the network is receiving the latest information about each requirement. By improving coordination, we prevent expensive rework and help contributors do their work more efficiently.

6. ACKNOWLEDGMENTS

Funding for this project is being provided by IBM Innovation Grant and Technology Fellowship Award, as well as NSERC.

7. REFERENCES

[1] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *CSCW '06: Proceedings of the 2006 conference on Computer-supported cooperative work, November 4–8*, 2006.

[2] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, 1988.

[3] D. E. Damian and D. Zowghi. Requirements engineering challenges in multi-site software development organizations. *Requirements Engineering Journal*, (8):149–160, 2003.

[4] K. Ehrlich and K. Chang. Leveraging expertise in global software teams: Going outside boundaries. In *International Conference on Global Software Engineering*, 2006. (To appear).

[5] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Softw.*, 16(5):63–70, 1999.

[6] L. G. Izquierdo. A case study of feature-based awareness in a commercial software team and implications for design of collaborative tools. Master's thesis, University of Victoria, 2006.

[7] I. Kwan, D. Damian, and M.-A. Storey. Visualizing a requirements-centred social network to maintain awareness within development teams. In *International Workshop on Requirements Engineering Visualization 2006 (REV'06), Minneapolis/St. Paul, Minnesota, USA, September 11–15*, 2006.

Supporting Cooperation Awareness in Common Information Spaces

Wolfgang Gräther
Fraunhofer FIT
Schloss Birlinghoven
53754 Sankt Augustin
+49 (0) 2241/14-2093
wolfgang.graether@fit.fraunhofer.de

Wolfgang Prinz
Fraunhofer FIT
Schloss Birlinghoven
53754 Sankt Augustin
+49 (0) 2241/14-2730
wolfgang.prinz@fit.fraunhofer.de

ABSTRACT

This position paper describes three different approaches to support cooperation awareness in common information spaces like shared file systems, Web sites, or shared workspaces. Smartmaps, activity map, and history map are graphical space-oriented visualizations. We present these visualizations and discuss their effect to collaboration.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: GUI; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work, Web-based interaction.

Keywords

Awareness, visualization, cooperation, common information spaces.

1. MOTIVATION

The importance of "awareness" for successful team work has been identified in many studies of workplaces [4]. Several models and applications have been developed and the requirement to provide awareness about activities and actions of others in a cooperative environment is part of CSCW engineering. Like in real world settings, situated action [7] requires awareness information about the working space in which the action takes place. This paper investigates the use of graphical space-oriented 2D visualizations to support awareness of presence and activities in common information spaces.

Smartmaps focuses on the provision of task-oriented awareness [6], i.e. they yield information about the state of artifacts and they peripherally inform cooperation partners about presence and ongoing activities of co-workers. Smartmaps show and enable access to all artifacts of a common information space and can therefore be used as alternative user interface even for large common information spaces with thousands of artifacts and tenths of co-workers.

The activity map focuses on the presentation of the level of activity for co-workers over a specified period of time. Co-workers become aware of ongoing activity and can access details about artifacts. The activity map enables self-control. The history map visualizes the complete event history for artifacts in BSCW shared workspaces. Co-workers can easily identify 'interesting' documents, i.e. accessed by all members of the workspace or read/written by certain co-workers.

In the following we will briefly describe each approach, illustrate how they can be integrated in a shared information space, and finally we will discuss their specific properties.

2. SMARTMAPS

Smartmaps are based on the treemap visualization technique [5] and represent the body of common artifacts in 2D graphic. Links from the 2D graphic to the artifacts are established allowing direct interaction with artifacts. The visualization eases traversing hierarchical structures. User actions on artifacts result in color-coding of the respective part of the Smartmap and therefore enables activity-based navigation. The presentation of user data helps in becoming aware of potential collaborators and advice giving experts.

2.1 Visualization

The Smartmap shown in Figure 1 presents activity information of a small common information space consisting of 42 artifacts. The 3 top-level folders have thick borders and contain 22, 9, and 11 artifacts, respectively. Folders on other levels in the hierarchy of the information space are not directly visible. All artifacts are represented as small rectangles with the same area; artifacts in the same folder are close to each other and ordered lexicographically.

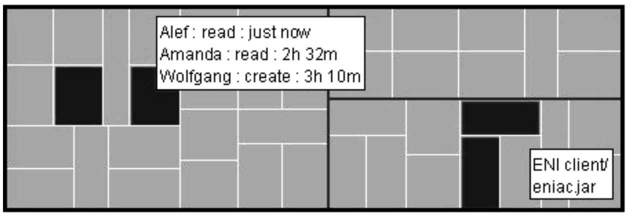


Figure 1. Small Smartmap: tooltips display location of artifacts and name of co-workers.

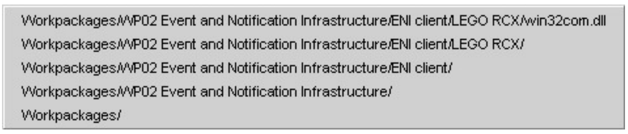
Highlighted rectangles, here in black, indicate user activity. The default presentation mode conveys the overall activity and their distribution in the information space. Tool tips, which are activated, when the user moves the mouse over the corresponding region, indicate location and name of the artifact. When the mouse is moved over a highlighted rectangle, then the tool tip presents the names, actions, and passed times of the last 3 users who have worked with the artifact. There are several parameters influencing the visualization.

For the duration of highlighting, we prefer 10 minutes for Smartmaps visualizing activities on Web sites and we favor duration of one day for BSCW shared workspaces and shared file systems. The latter duration enables users to see at a glance an overview of what has happened in the common information space during the last 24 hours.

2.2 Interaction

Smartmaps support not only the visualization of activity information in common information spaces, but they also ease the navigation in structured information spaces and provide access to the artifacts. There are a lot of interaction possibilities on Smartmaps:

- Moving the mouse over the Smartmaps presents either the location and the name of the artifact or information about recent user activities,
- a mouse click presents the pathname of the artifact in the status bar of the browser window,
- a shift-mouse click opens the artifact itself,
- a control-mouse click opens the enclosing folder,
- a control-right-mouse click presents a popup menu to open the artifact, the enclosing folder, and further enclosing folders up to the top-level folder (see Figure 2),
- a right-mouse click shows a popup menu to set visualization parameters and to access further functions like help and about.



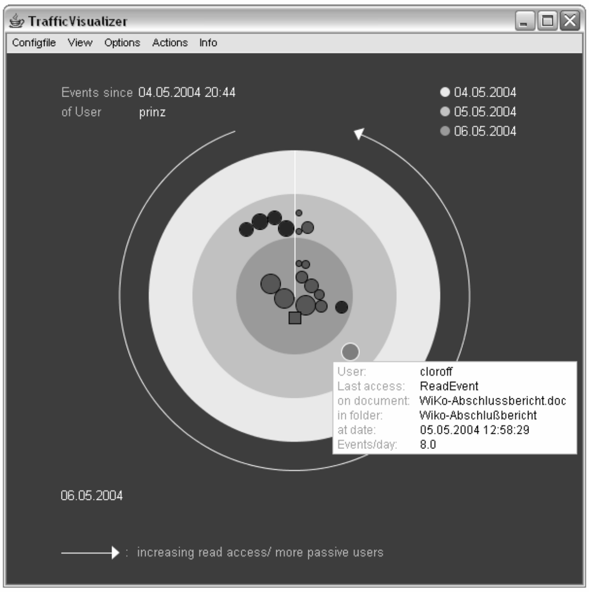
Workpackages/WP02 Event and Notification Infrastructure/ENI client/LEGO RCX/win32com.dll
Workpackages/WP02 Event and Notification Infrastructure/ENI client/LEGO RCX/
Workpackages/WP02 Event and Notification Infrastructure/ENI client/
Workpackages/WP02 Event and Notification Infrastructure/
Workpackages/

Figure 2. Popup menu to access directly different parts of the common information space.

3. ACTIVITY MAP

The activity map is inspired from the work on Babble [2]. It visualizes user activity in BSCW shared workspaces according to passed time of the last action and to a participation measure. The user is placed according to his/her activeness in 'the circle' of the other co-workers. This presentation of activity enables self-control and shows a detailed overview of workspace participation to workspace managers.

The activity map shown in Figure 3 presents for user 'prinz' his activity and the activity of 18 co-workers. The user 'prinz' is represented as a square; co-workers are visualized as circles in different colors. The size of the user representation is linear to the number of actions on the artifacts (documents) in the BSCW shared workspace, i.e. larger circles indicate more actions.



Events since 04.05.2004 20:44
of User prinz

User: cloroff
Last access: ReadEvent
on document: WiKo-Abschlussbericht.doc
in folder: Wiko-Abschlußbericht
at date: 05.05.2004 12:58:29
Events/day: 8.0

Figure 3. Activity map: tooltip displays detailed info about co-worker, action and artifact.

The main part of the visualization is 3 concentric circles drawn in different colors representing different time periods. These periods can be chosen in a settings dialog. In the example above these are three consecutive days: 6th of May (inner circle), 5th of May (middle circle) and 4th of May (outer circle) in 2004. The users circle is placed according to the date of her/his last action. For example, the circle of 'cloroff', directly above the tool tip, indicates that this user was active 1 p.m. at the 5th of May 2004.

For the presentation of user activity we developed a simple participation measure. All events were classified either into the class active or passive. Read events fall into the class passive, all other events like write, delete, or move fall into the class active. The ratio, i.e. the size of the class active divided by the number of all events, indicates the activeness of a user.

The users are placed according to their activeness in the visualization of the activity map. The level of activeness is highest at 12 o'clock and gets lower anticlockwise. For example, the user 'cloroff' and 'prinz' both have a medium activeness. Lurkers, i.e. really passive users, can be found in the activity map directly to the right of 12 o'clock. Figure 3 shows, of course, a less number of active than passive users.

Tool tips are activated, when users move the mouse over a circle (represented user). The name of the co-worker, the name and date of the last action, the name of the artifact and the name of the enclosing folder is shown in the tool tip. The activity map offers a search function for users, a replay function and a few more settings.

Crucial for the usefulness of the visualization is the setting of the time periods for the three concentric circles. For busy BSCW shared workspaces we recommend three consecutive days or as inner circle the current day, as middle circle the last two days and as outer circle the other five days, so that a complete week is covered.

4. HISTORY MAP

The history map visualizes in a slice-and-dice manner all user actions on artifacts in BSCW shared workspaces. This visualization enlarges artifacts, which have been often accessed. For every artifact the share of actions for each user is displayed. This presentation of activity enables co-workers, for example, to access only often accessed (interesting) artifacts. Social navigation is a second characteristic of the history map.

The history flow visualization [8] is related to the history map but focuses on the presentation of changes in common documents over time.

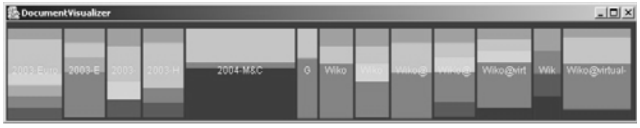


Figure 4. History map: tooltips display name of artifacts or name of co-workers.

The history map shown in Figure 4 presents the actions of a BSCW shared workspace. The workspace contains 12 documents and 1 subfolder (2004-M&C) with 5 more documents (invisible). Note that only actions on artifacts and on artifacts in direct subfolders are presented. This subfolder-constraint keeps the visualization and interpretation simple.

The artifacts and subfolders are presented in the history map as slices. The thickness of the slices depends on the number of actions, which have been taken place on the artifact. Very often accessed artifacts are represented as broad slice, less often artifacts are represented as small slice. The ordering of the artifacts is lexicographically.

The slice for the artifacts is further divided into dices (horizontal rectangles) according to the share of actions the respective co-worker has. These areas have specific colors dependant on the co-workers. The size (height) of the area is proportional to the activity of the user. Of course, color-code for users and ordering of the user names remain the same within a BSCW shared workspace.

Tool tips are activated, when users move the mouse over the history map. Usually the names of the co-workers are displayed. If the mouse is moved over the partly visible name of the artifact, then the complete name is shown in a tool tip.

The history map is useful for small common information spaces with less than 50 artifacts and less than 20 co-workers accessing these.

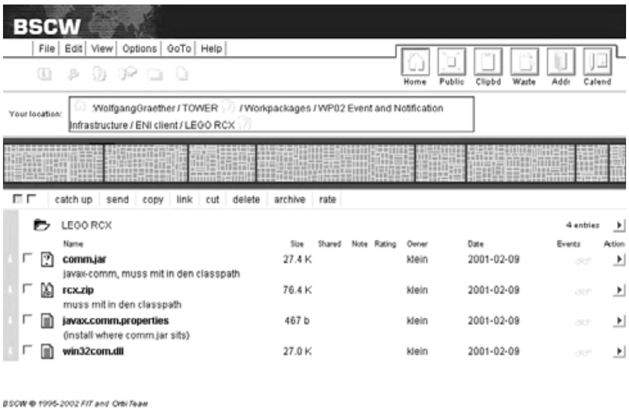
5. SMARTMAPS AS PART OF BSCW SHARED WORKSPACES

In this chapter we explain only the application of Smartmaps integrated into BSCW shared workspaces. The activity map as well as the history map is also designed for being integrated into common information spaces, which makes them applicable both for visualization of common information spaces and interaction with common artifacts.

The BSCW shared workspace system [1] is a Web-based groupware tool based on the notion of shared workspaces, which may contain various kinds of objects such as documents, tables, graphics, spreadsheets or links to other Web pages. Folders are used to group the artifacts and the hierarchy of folders constitutes the shared workspace. Workspaces can be set up, members can be invited and objects can be stored, managed, edited or downloaded with any Web browser. There are many other services available, for example, discussion forums. Asynchronous as well as synchronous collaboration is supported by BSCW.

The BSCW system offers a user customizable area, the banner described in HTML, which partly changes the display of workspace and folders. Typical banners are: images or headlines according to workspace content, project logos, Web cam images, and even applets. The banner is inherited through the hierarchy of folders in the shared workspace.

Smartmaps are realized as Java-Applets and in contrast to the activity map suitable for the integration into BSCW shared workspaces. Figure 5 shows a Smartmap integrated into a BSCW shared workspace with more than 1000 artifacts. At every place in the workspace the Smartmaps is shown and augments the usual list mode presentation of the shared objects. The actual position of the user in the hierarchy of the shared workspace is indicated by an orange rectangle.



Your location: :WolfgangGraether / TOWER / Workpackages / WP02 Event and Notification Infrastructure / ENI client / LEGO RCX

LEGO RCX 4 entries

Name	Size	Shared	Note	Rating	Owner	Date	Events	Action
comm.jar javax-comm, muss mit in den classpath	27.4 K				klein	2001-02-09		
rcx.zip muss mit in den classpath	76.4 K				klein	2001-02-09		
javax.comm.properties (install where comm.jar sits)	467 b				klein	2001-02-09		
win32com.dll	27.0 K				klein	2001-02-09		

BSCW © 1995-2002 FIT and Orbi Team

Figure 5. Smartmaps in a BSCW shared workspace.

We have integrated Smartmaps into several large project workspaces and we could observe that users quickly apply the Smartmap. For example, they first check the places for which activity of other co-workers is indicated. They move the mouse to the corresponding highlighted rectangle to see who acted on the artifact, what kind of action took place, and when the action happened. Then, often that link is followed and the corresponding folder or the object is opened.

Smartmaps in shared workspaces complement awareness information which is already available at the shared artifacts, but visible only in the current work situation, with awareness information from other parts of the overall working context of the user. An even larger context could be built, when several Smartmaps, representing different workspaces, are put together.

6. DISCUSSION

Table 1 compares the three approaches, which will be applied and further evaluated in different software development projects in the context of the SAGE project. SAGE [3] aims at the development of new solutions for self-organized cooperative task management and group awareness for the coordination of distributed software development processes.

Table 1. Comparison of Smartmap, activity map, and history map

Aspect	Smartmap	Activity Map	History Map
Purpose	Visualization of large common information spaces combined with object-based activity information.	Visualization of user-based activity information to provide an overview of the group activity.	Visualization of small to medium sized common information spaces combined with details about object and user specific activities.
Application	Activity-based navigation of large common information spaces. Fast overview and indication of activity hot spots: "In which areas of the common information space did something happen?"	Overview on group activity. "Who was active and when was the last action?", "Have users been more actively or passively involved?"	Overview on object specific activities. "What are the objects with most activities?", "Which objects have been used by most users?"
Represented artifact	Objects (documents)	Users	Objects (documents)
Mouse over action	Object information and user activity	Detailed event information	User information
User interaction	Browsing the common information space.	Access to further event details and direct access to objects.	Access to further event details and direct access to objects.
Use of space …	to provide an overview of the common information space.	to show the relation between active and passive users as well as the time since last activity.	to show the relation between the number of events on different objects.
Use of color …	to indicate activity on objects.	to indicate the type of the last activity.	to distinguish between users.
Supported type of awareness	Task-oriented	Social	Task-oriented and social

7. ACKNOWLEDGEMENT

We would like to thank Christian Stein, who implemented the activity map and the history map. We would also like to thank our colleagues of the TOWER and SAGE project teams. The TOWER project was partly funded through the IST program IST-10846 of the European Commission. The SAGE project is partly funded through the German Federal Ministry of Education and Research.

8. REFERENCES

[1] Appelt, W. WWW Based Collaboration with the BSCW System. In *Proceedings of 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99)* (Milovy, Czech Republic, 1999). Springer Verlag, 1999, 66-78.

[2] Bradner, E. The Adoption and Use of 'BABBLE': A Field Study of Chat in the Workplace. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)* (Copenhagen, The Netherlands, September 12-16, 1999). Kluwer A.P., 1999, 139-158.

[3] Gräther, W., Koch, T., Lemburg, C., Manhart, P. (2006, forthcoming) SAGE: Self-organized cooperative task management and group awareness for the coordination of distributed software development processes.

[4] Grinter, R. (1997). From Workplace to Development: What Have We Learned So Far and Where Do We Go? In *Proceedings of GROUP'97.* ACM Press, 1997, 231-240.

[5] Johnson, B., Shneiderman, B. Treemaps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of IEEE Visualization'91,* San Diego 1991. IEEE Computer Society Press, 1991.

[6] Prinz, W. An Awareness Environment for Cooperative Settings. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)* (Copenhagen, The Netherlands, September 12-16, 1999). Kluwer Academic Publishers, 1999, 391-410.

[7] Suchmann, L. A. *Plans and situated actions: The problem of human-machine communication.* Cambridge University Press, 1987.

[8] Viégas, F.B., Wattenberg, M. Dave, K. Studying Cooperation and Conflict between Authors with history flow Visualization. In *Proceedings of CHI 2004.* ACM Press New York, NY, USA 1999, 575-582.

Managing Complexity in Collaborative Software Development: On the Limits of Modularity

Marcelo Cataldo[1] Matthew Bass[1] James D. Herbsleb[1] Len Bass[2]
[1] Institute for Software Research International
[2] Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
mcataldo@cs.cmu.edu mbt@sei.cmu.edu jdh@cs.cmu.edu ljb@sei.cmu.edu

ABSTRACT

The identification and management of dynamic dependencies between components of software systems is a constant challenge for software development organizations. In this paper, we discuss 4 case studies that exemplify the complexity of identifying and managing dependencies in a global software development project. The uncertainty of the interfaces and the nature of the dependency are key factors in determining the need for communication and coordination. Interestingly, we encountered cases where even simple interfaces between modules developed by remote teams create coordination breakdown and development problems, raising questions regarding the effectiveness of traditional mechanisms to divide work, such as modularization.

1. INTRODUCTION

In the system design literature, it has long been speculated that the structure of a product inevitably resembles the structure of the organization that designs it [2]. In Conway's original formulation, which has come to be known informally as Conway's Law, he reasoned that coordinating product design decisions requires communication among the engineers making those decisions. If everyone needs to talk to everyone, the communication overhead does not scale well for projects of any size. Therefore, products must be split into components, with limited technical dependencies among them, and each component assigned to a single team. Conway proposed that the component structure and organizational structure stand in a homomorphic relation, in that more than one component can be assigned to a team, but a component must be assigned to a single team. Parnas took a similar view talking specifically about software, in his classic paper on modular design, in which he considered modules to be work items instead of a collection of subprograms [9].

A similar argument has been proposed in the strategic management literature. Baldwin and Clark [1] argued that modularization makes complexity manageable, enables parallel work and tolerates uncertainty. The design decisions are hidden within the modules which communicate through standard interfaces, then, modularization adds value by allowing independent experimentation of modules and substitution [1]. Although Baldwin and Clark's analysis is at the industry level, it is clear that their view aligns with Conway's idea that one or more modules can be assigned to one organizational unit and work can be conducted almost independently of others.

Both theoretical arguments rely on the assumptions that the interfaces between modules are stable and well defined, consequently, minimal communication between the organizational units involved is necessary. However, in large and complex software systems dependencies range from simple syntactic relationships (e.g. a function call) to more complex and difficult to identify dependencies such as a semantic dependency. Moreover, modifications to the software introduce constraints that might establish new dependencies among the various parts of the system, modify existing ones or even eliminate dependencies. Failure to discover the changes in coordination needs might have a profound impact on the quality of the product and productivity [3, 6, 7]. This dynamic nature of task dependencies in software development is a key problem overlooked by the modularization perspective.

The identification and management of dynamic dependencies between components of software systems is a constant challenge for software development organizations. As geographically distributed software development projects become pervasive, understanding the processes, tools and organizational factors that matter the most for identification and management of dynamic dependencies is an important research endeavor. In this paper, we present a preliminary analysis of critical incidents in a global software development project. The Global Studio Project [8], sponsored by Siemens Research, is a *test-bed* where groups of graduate students from several universities work on project that simulates a real-life software development effort. We discuss 4 case studies that exemplify the complexity of identifying and managing dependencies. We also present cases where even simple interfaces between modules developed by remote teams create coordination breakdown and development problems. Finally, we argue that the uncertainty of the interface as well as the nature of the dependency calls for different collection of processes, tools and organizational structure to provide the necessary means to identify and manage dependencies.

Computer Supported Cooperative Work '06, November 4–8, 2006, Banff, Alberta, Canada.

2. STUDIO PROJECT

The Global Studio Project (GSP) [8] was established by Siemens Corporate Research (SCR) as a test bed to gain better understanding of the issues associated with global software development. The project simulates a real world geographically distributed project by using student teams to develop software. The students participate in the GSP as part of their academic curriculum and they operate in academic environments at universities in Ireland, Brazil, Germany, India and the United States. These students are pursuing their masters or diplomas (equivalent to masters) in software engineering or associated fields. The student groups had no previous experience of working together.

The system developed in the GSP, called MSLite system, is to be a unified management station for building automation systems such as heating ventilation and air conditioning (HVAC), access control, and lighting that will allow a facility manager to operate such systems.

The GSP is organized in a two-level hierarchical structure with a central team located at SCR that is responsible for specifying requirements, software architecture and some aspects of design, system test, integration, project management and defining processes for code submission, testing, and communication. The remote teams are responsible for design, development and unit tests for particular code modules or sub-systems defined by the central team. The central team has a Supplier Manager (SM) for each remote team, whose responsibility is to mange the interactions between the central team and the remote team which also has its local Supplier Manager.

The central team used the architecture documentation to identify dependencies among components and then generate a design structure matrix (DSM). Following an analysis similar to Baldwin & Clark's [1], the DSM was used to identify the set of tasks to be assigned to each remote team that would minimize the dependencies and consequently, minimize the coordination requirements between remote teams.

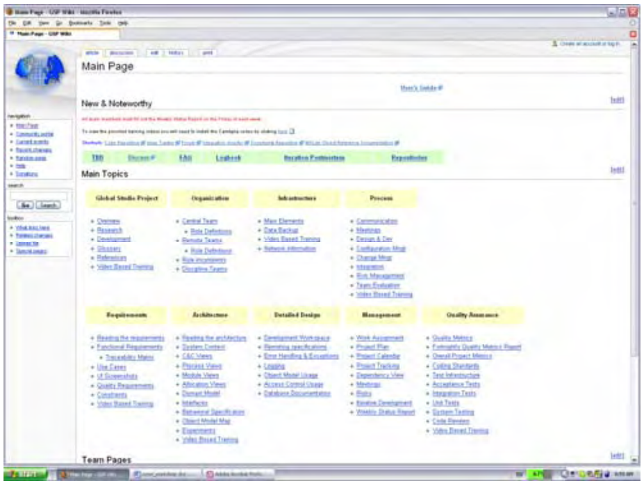


Figure 1: GSP main Wiki page

Following best practices from the open source community [5], the GSP provided a wiki web portal (figure 1) which gave users access to a host of tools and information about the project such as architectural documentation, information about the teams involved in the project, processes, version control system, discussion forums, defect tracking system and a daily build system. The central team established processes for communications and meeting, design and development, configuration and change management and integration of the code.

Communication processes emphasize the usage of email to interact with the SM. Also developers were encouraged to use the discussion forums for technical discussions with other teams. All remote teams had weekly status meetings with the central team's SM and all developers were expected to post their weekly progress on the web portal. The development tasks were divided in 6 iterations of 8 weeks each. At the end of an iteration, each remote team had deliverables that include specifications, functioning code and unit tests. In some cases, deliverables were due at the half point in the iteration.

Modifications to the software code were managed by a central version control system. All teams were encouraged to submit their changes regularly to the central repository. A daily build system would make sure that the code compiled correctly and that the appropriate set of unit tests ran successfully. If a problem was encountered, the build system would send email to the central team and to the team that made the last submission to the version control system, who became responsible for resolving the problem. If the issue persisted overnight, the central team would revert the changes the following morning.

3. PROBLEMS MANAGING DEPENDENCIES

The initial data collection was done using an approach similar to the critical incidents technique [4]. We met with the members of the central team and we asked them to identify events during the life of the project that were representative of important problems in coordination. The central teams members were also asked to provide background information regarding the events they recalled. We then compiled data associated with the incidents from numerous sources such as technical documentation, version control system, project plan, meetings minutes, weekly status reports of the developers, discussion forum, defects database, email archive and social network survey. The following sections describe these events in detail.

3.1 Event I: Change in a design specification.

The team in Ireland was responsible for task A in iteration 2. The team in India was responsible for task B in iteration 3. Task A consisted of designing several object classes and specifying the properties and methods of those classes. Task B implemented a property editor that used the object classes defined in Task A. The developers involved in both tasks participated in three different discussion forums focused on the technical details on the implementation for task A. All the technical details of task A were not captured correctly in the design specification document. This mistake led to a serious mismatch between the contents of the documentation and the actual implementation, a perennial problem in software development.

The Indian team worked on task B guided primarily by the contents of the design specification which led to integration problems when they submitted their changes into the version control system. Our analysis suggests that some members of the Indian team were reluctant to make full use to the version control system. The time zone difference between India and the US EST (the location of the central team), meant that around the time the Indian team was leaving for the night, the central team would be starting their workday. If any submissions to the code repository resulted in a broken build, the Indian team would not have the opportunity to fix the problem before the central team would revert the changes. Hence, the Indian team tended to rely a lot more on documentation and also tended to make less frequent but much larger changes to the code.

This incident highlights several interesting issues. First, despite the availability of numerous communication tools (e.g. email, discussion forums, and defect reports), our analysis indicated that very little lateral communication between the teams took place. Exchange of information between the teams was limited to discussion forums early in the life of task A. We think an important factor is the central management approach tended to emphasize information flow through the central team. Secondly, incomplete or incorrect documentation can have a major negative in productivity and quality in the context of geographically distributed software development. Finally, the case exemplifies the role that processes play in shaping the behavior of certain developers and, ultimately, the coordination of activities. Although nightly builds are considered an effective practice, the consequences that resulted from breaking the build diminished the value of this process.

3.2 Event II: Modification of a major interface.

A team in a US university was responsible for implementing a data access interface that all other components of the system depended on it. One of the requirements of the data access module was to uniquely identify instantiated objects. The developer responsible for the task evaluated the original design specification done by the central team and determined that the interface needed to be modified in order to satisfy the requirement to generate unique object identifiers. As required by the design and development processes, the developer sent a proposal for the design change to the central team. However, that took place two days before the deliverables were due. Since there was no reply from the central team, the developer submitted the modifications proposed in his design to the version control system two days later. These actions resulted in some major modifications in various parts of the system causing delays and frustration on the other remote teams. The following email trace shows all the information exchanged between the developer team (named Team X) and the central team regarding this issue:

To: ALL TEAMS
Sent: 3/27/2006 3:28 PM
Subject: Access Control Modifications
All,
As you may already have noticed the Access Control component underwent some changes to incorporate the inclusion of an AccessControlResultSet. Information on the Result Set may be found at . . . In order to integrate the changes, some teams' code may have required slight modifications which were carried out by the team X. These changes were authorized by the central team and were essential for successful server-side integration. Please review your code to ensure all changes were satisfactory.
Thanks and best regards,
Central Team SM

-----Original Message-----
From: Team X
Sent: Friday, March 24, 2006 11:18 AM
To: Central team SM
Subject: AccessControlResultSet design
All,
Attached is the detailed design of the AccessControlResultSet and the updated detailed design of the AccessControl component. Please review and have comments to me before Mondays teleconference meeting. best regards,
Developer
-----Original Message-----
From: Central Team SM
To: Team X
Sent: 3/23/2006 11:31 AM
Subject: FSS .NET Remoting Failing test
Thanks. Can you update me on your progress?

This incident is a good example of several related problems. First, we have a change to an interface, a syntactic dependency, that becomes a major problem because the interface in used many times in all the components of the system. Furthermore, the semantic of the functionality also changed (generate and return a unique identifier), augmenting the scope of the change. Syntactic dependencies tend to be misleading because they are typically considered simple issues. This example shows that certain types of dependencies (e.g. numerous modules depend on the same interface that returns a critical data type) require a lot more attention than other, particularly, during the design phase. Early identification of such types of dependencies can also help focus the efforts of management in the most critical aspects of the project. Another interesting issue highlighted by this incident is the impact of lack of contextual information and conflict. After the central team announced that major changes would take place in the code, several teams expressed frustration with development team X because the changes to be made would delay their current work. However, none of the remote teams had a complete view and understanding of why the changes were necessary.

3.3 Event III: Circular dependency between components.

One US university team was responsible for task A due at the midpoint of iteration 3. Another team from a different US university was responsible for task B due at the end of iteration 3 of the project. The information in discussion forums and email indicated that the teams had extensive exchange of technical information associated with interfaces developed as part of task A and that the component developed as part of task B would depend on. These interfaces represented a case of syntactic dependencies between two components (A → B). Upon completion of task A, the central team and the development team did a detailed code review and the modifications were approved.

Unfortunately, the architectural documentation also revealed a more complex semantic dependency through a publisher/subscriber mechanism that would need to be resolved as part of task B (B → A). This dependency went undetected during the code review and this mistake resulted in major modifications to the component developed in task A during the execution of task B. Code reviews are well known practices in software engineering. Although code reviews are commonplace, they do not typically involve an analysis of dependencies to determine appropriate set of developers to participate in the review [10]. This incident shows how important it is to identify all the dependencies among components in order to have the code reviewed by the relevant developers. Distance and the central management approach tend to augment the impact of this type of mistake because impromptu and lateral communication are limited.

3.4 Event IV: Significant delays in the implementation of a complex module.

The development of a low priority but complex component was originally scheduled for iteration 3 to be done by the Irish team. Modules developed in iteration 5 were dependent on this component. The Irish team was not able to finish the design of the component so the design and development was re-scheduled for iteration 4 to be done by the group in Brazil. Shortly after the Brazilian team did the preliminary analysis and estimated the effort to complete the component, the Supplier Manager for the Brazilian team got sick and communications with the central team dropped almost to a halt. Ultimately, the task had to be re-scheduled again for iteration 5 and this time one of the US universities teams would be responsible.

The delay in the implementation of this particular component changed the dynamics of the coordination required between the teams. A fairly loose coupling (Team A implements the module, then during a subsequent iteration, Team B uses it) became a very tight coupling (implementation and use are concurrent). The development teams involved are located one in the US and one in Germany. The need for fluid exchange of information and tight coordination are critical as the design decisions made by one team could have implications in the other team's development efforts. Our qualitative analysis revealed that little lateral communication is taking place between the remote teams. Moreover, numerous modifications to the code had to be reverted because the changes broke the build. This situation escalated to a clear point of frustration as the following message in a modification to the code indicates:

```
r1901 | Central team SM | 2006-08-10 14:04:44 -0400
Changed paths:
  M /MSLite/MSLite.Rules/ConditionEvaluation/Evaluator.cs
 ….
  D /MSLite/MSLite.Tests.Rules/RuleEngineTests.cs
Reverting repository revisions 1898 and 1897 by "Developer A"
in attempt to successfully build integration server – AGAIN
```

4. DISCUSSION

The cases highlight the importance of certain properties of the software product to be developed. First, nature of the dependency matters. Complex semantic relationships cause coordination problems (event III) even after well established quality assurance processes took place. However, and possibly more important, is the fact that coordination problems also arise with simple syntactic interfaces as it is illustrated by event I.

Uncertainty of the interfaces is another important factor to consider because uncertainty opens the door for potentially serious coordination problems. Although the event didn't explicitly provide an example of such a situation, the perception of simplicity could lead to similar problematic conditions. In event II, a major re-development effort took place because of a design that did not contemplate all the requirements. An interface that was originally considered "straightforward" needed to be re-architected and consequently numerous parts of the system had to be modified.

In sum, the collection of events presented here suggests that the coordination problems encountered in geographically distributed software development project depend on an intricate relationship of several factors. First, elements that influence how closely-coupled the work is, such as complexity and uncertainty of the interfaces, as well as whether the work is carried out sequentially or concurrently. Secondly, factors that influence the ability to communicate and coordinate, such as geographic separation, whether communication is direct or through an intermediary, and the quality of documentation play a significant role. Finally, other organizational factors such as processes, structure and goal alignment are important mediators as well.

5. ACKNOWLEDGMENTS

The authors gratefully acknowledge support by NSF grant IIS-0534656, as well as support from the Software Industry Center and its sponsors, particularly the Alfred P. Sloan Foundation. The authors would also like to thank the support of the Software Engineering Institute and the US Department of Defense. Part of this work was also supported by the National Science Foundation IGERT program 9972762 and the center for Computational Analysis of Social and Organizational Systems at Carnegie Mellon University.

6. REFERENCES

[1] Baldwin, C.Y. and Clark, K.B. *Design Rules: The Power of Modularity*. MIT Press, 2000.

[2] Conway, M.E. How do committees invent? *Datamation*, 14, 5 (1968), 28-31.

[3] Curtis, B., Kransner, H. and Iscoe, N. A field study of software design process for large systems. *Comm. ACM,* 31, 11 (Nov. 1988), 1268-1287.

[4] Flanagan, J.C. The Critical Incident Technique. *Psychological Bulletin*, 51, 4 (July 1954).

[5] Halloran, T. and Scherlis, W. High quality and open source practices. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, Florida, May 2002.

[6] Herbsleb, J.D. and Mockus, A. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Trans. on Soft. Eng.*, 29, 6 (2003), 481-494.

[7] Kraut, R.E. and Streeter, L.A. Coordination in Software Development. *Comm. ACM,* 38, 3 (Mar. 1995), 69-81.

[8] Mullick, N. et al. Siemens Global Studio Project: Experiences Adopting a GSD Infrastructure. In *Proceedings of the International Conference on Global Software Engineering*, Florianopolis, Brazil, September 2006.

[9] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM,* 15, 12 (1972), 1053-1058.

[10] Wiegers, K.E. Peer Reviews in Software: A Practical Guide. Addison-Wesley, 2001.

WYSIWYN: Using Task Focus to Ease Collaboration

Mik Kersten, Rob Elves and Gail C. Murphy
Department of Computer Science
University of British Columbia
{beatmik, relves, murphy}@cs.ubc.ca

ABSTRACT

WYSIWYG (What You See Is What You Get) tools changed how knowledge workers and others produce and collaborate on documents. Our Mylar project is showing how WYSIWYN[1](What You See Is What You Need) tools can change how programmers work and interact. As a programmer works on a task, Mylar builds a context for the task that captures which resources are interesting to complete the task. These task contexts can be used to focus the user interface with which a programmer works, reducing the overload of information the programmer usually experiences. Sharing task contexts can also focus collaborative programming activities, making it easy to show a team member how a bug was solved. We have shown that Mylar makes programmers more productive in a field study of 16 programmers using Mylar for several weeks. In this position paper, we provide an overview of Mylar and discuss some further ways in which WYSIWYN may improve programmers' use of integrated development environments.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments — *integrated environments, programmer workbench*.

General Terms

Design, Human Factors

Keywords

Task-based interaction, Degree-of-interest, Focused user interfaces

1. INTRODUCTION

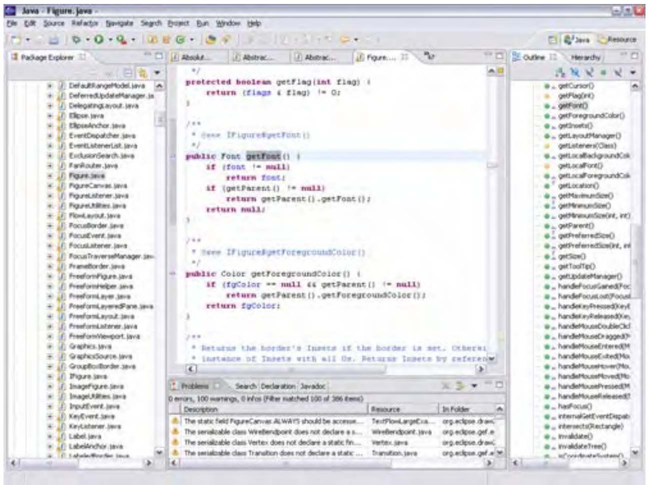
WYSIWYG (What You See Is What You Get) had a fundamental impact on the productivity of knowledge workers. For example, these tools made it possible for organizations to create professional quality newsletters that keep current and past members of an organization aware of events happening at the organization. As another example, these tools changed the work performed by administrative staff as individuals within an organization became responsible for formatting letters and documents they wrote.

In our Mylar project[2], we are investigating how WYSIWYN (What You See Is What You *Need*) tooling can change how individual programmers work and how those programmers work together. WYSIWYN tooling *focuses* the information presented to programmers on just the information that he or she needs to complete individual tasks and to collaborate with others on the tasks associated with a project.

In this position paper, we provide a brief overview of Mylar, describing how it provides WYSIWYN support to both an individual programmer and to teams of programmers. We also briefly discuss further ways in which WYSIWYN support could be added to Integrated Development Environments (IDEs) to facilitate collaboration.

2. MYLAR

Many programmers spend much of their time working in an IDE. The trend in IDEs has been to add more and more features that are able to quickly display more and more information about the system to the programmers. Figure 1 shows a screenshot of the typical views facing a Java programmer working in the Eclipse IDE[3]: each view is populated with numerous program elements, requiring the developer to scroll and search to find the elements needed for the task-at-hand. The result of making it possible to easily display a large subset of a system's artifacts is that programmers spend more time looking for elements they need to complete a task than they spend actually working with those elements. Unless they are systematic in looking for the elements of interest, they can suffer from inattention blindness, missing relevant items that may appear on the screen accidentally [6]. This problem is exacerbated by two aspects of a programmer's work: 1) a programmer switches between tasks frequently [2] and 2) a programmer often collaborates with other team members.



```
*/
protected boolean getFlag(int flag) {
    return (flags & flag) != 0;
}

/**
 * @see IFigure#getFont()
 */
public Font getFont() {
    if (font != null)
        return font;
    if (getParent() != null)
        return getParent().getFont();
    return null;
}

/**
 * @see IFigure#getForegroundColor()
 */
public Color getForegroundColor() {
    if (fgColor == null && getParent() != null)
        return getParent().getForegroundColor();
    return fgColor;
}

/**
 * Returns the border's Insets if the border is set. Otherwi
 * instance of Insets with all 0s. Returns Insets by referen
```

0 errors, 100 warnings, 0 infos (Filter matched 100 of 386 items)

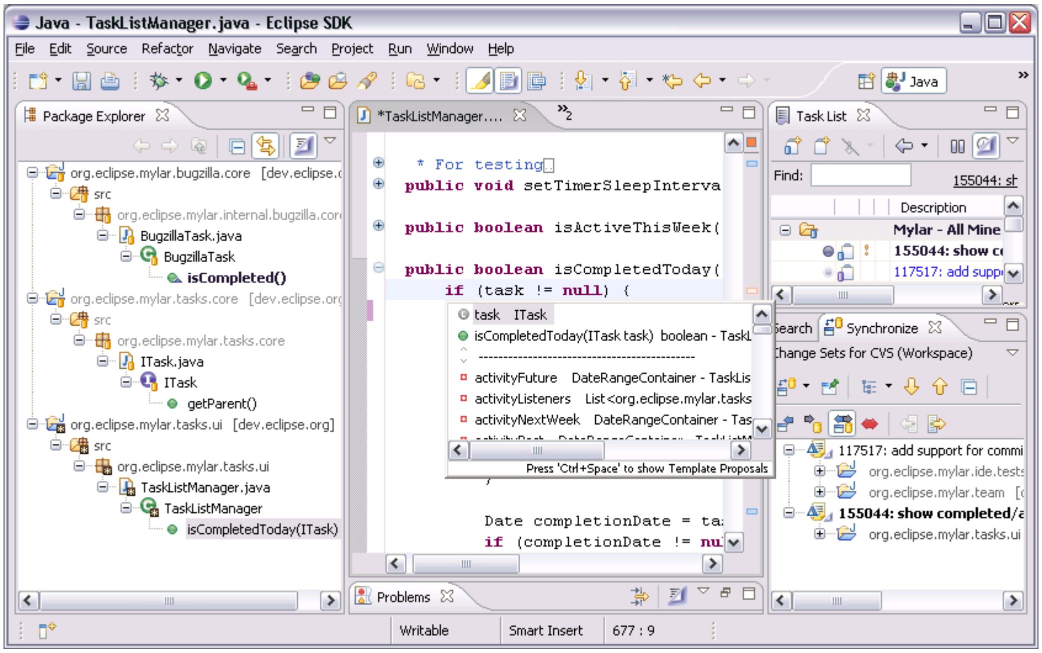
Description	Resource	In Folder
The static field FigureCanvas.ALWAYS should be accesse...	TextFlowLargeEx...	org.eclipse.draw
The serializable class WireBendpoint does not declare a s...	WireBendpoint.java	org.eclipse.gef.e
The serializable class Vertex does not declare a static fin...	Vertex.java	org.eclipse.draw
The serializable class Transition does not declare a static ...	Transition.java	org.eclipse.gef.e

Figure 1 An Overloaded IDE Workspace

[1] www.usabilityfirst.com/glossary/term_1070.txl, verified on 06/10/06

[2] www.eclipse.org/mylar, verified on 12/09/06.

[3] www.eclipse.org, verified on 12/09/06.


Java - TaskListManager.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
org.eclipse.mylar.bugzilla.core [dev.eclipse.c
src
org.eclipse.mylar.internal.bugzilla.core
BugzillaTask.java
BugzillaTask
isCompleted()
org.eclipse.mylar.tasks.core [dev.eclipse.org
src
org.eclipse.mylar.tasks.core
ITask.java
ITask
getParent()
org.eclipse.mylar.tasks.ui [dev.eclipse.org]
src
org.eclipse.mylar.tasks.ui
TaskListManager.java
TaskListManager
isCompletedToday(ITask)
*TaskListManager....
Task List
Find:
155044: sh
Description
Mylar - All Mine
155044: show co
117517: add supp
Search
Synchronize
Change Sets for CVS (Workspace)
117517: add support for commi
org.eclipse.mylar.ide.tests
org.eclipse.mylar.team [c
155044: show completed/a
org.eclipse.mylar.tasks.ui
Problems
Writable
Smart Insert
677 : 9

```
 * For testing
public void setTimerSleepInterva

public boolean isActiveThisWeek(

public boolean isCompletedToday(
    if (task != null) {

    Date completionDate = ta
    if (completionDate != nu
```

```
task  ITask
isCompletedToday(ITask task)  boolean - TaskL
------------------------------------------
activityFuture   DateRangeContainer - TaskLis
activityListeners   List<org.eclipse.mylar.tasks
activityNextWeek   DateRangeContainer - Tas
Press 'Ctrl+Space' to show Template Proposals
```

Figure 2: A Task-Focused Workspace with Mylar

Our Mylar tool addresses the overload problem by focusing the information presented in the IDE around tasks. For instance, a programmer can see just the information needed to work on a particular task. One team member can see the status of tasks (issues) being worked on within the team. When one programmer collaborates with another, the first can easily share just the information that he or she considered and changed while working on the task with the team member.

2.1 Mylar for an individual programmer

A programmer working with Mylar indicates the current task by indicating the issue on which they are working from an issue repository.[4] Mylar's Task List supports queries over an issue repository. The Task List in the upper right view of Figure 2 shows tasks resulting from a query over a Bugzilla repository[5]. To indicate a particular issue is the current task, the programmer activates the issue by pressing the small round button at the left of a particular issue's name. In this case, the programmer has activated the issue #155044.

Once a task is activated, Mylar begins to monitor the resources (program elements and other information) a programmer accesses to perform the task. With this information, Mylar builds a model of which resources are important for the task. This model assigns a degree-of-interest to each resource based on the number of edits and selections of the resource [3,4]. We refer to the degree-of-interest model for a task as a task context. A task's context can be used as input to several operations. For instance, the context can be used to highlight the information presented or it can be used to filter uninteresting information. The left side of Figure 2 shows the Package Explorer (a view displaying the hierarchical containment hierarchy of the software) filtered to show only resources interesting for the current task. Comparing this to the workspace shown in Figure 1, we see that activation of the task focuses the views in the IDE to just what the programmer needs at present. Mylar retains the context for a task between activations of the task. When a programmer returns to work on a task, the programmer simply needs to reactivate the task and Mylar reloads the context displaying only the information needed for the task.

To investigate whether Mylar does enable programmers to spend more time working with resources than looking for them, we performed a field study in which 16 programmers used Mylar for their daily work for a period of several weeks. We benchmarked the activity of these programmers in Eclipse prior to providing them with Mylar. With statistical significance, we found that these programmers spent more time editing code than navigating it when using Mylar [4].

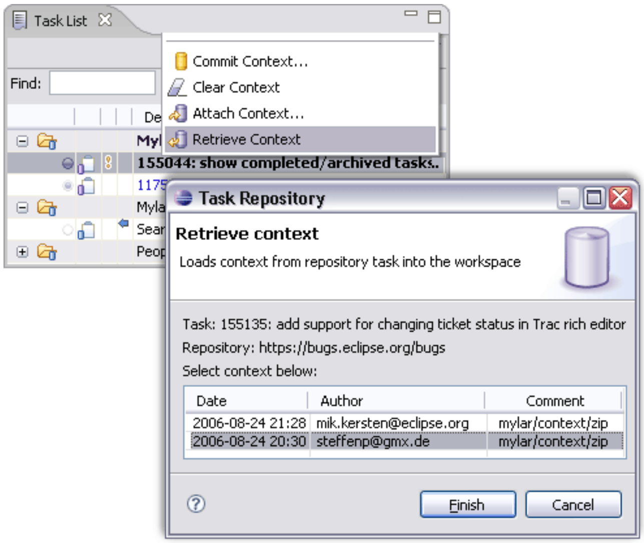
2.2 Mylar for collaborating programmers

Two or more programmers often end up working on the same task. This work may occur by the programmers huddling around the same workspace, it may occur by the programmers sequentially passing the task back and forth with one programmer making some progress and then passing it to another who has expertise in a different area, or it may occur separated in time with one programmer revisiting a previously completed task because of a newly reported bug or a desired enhancement.

Mylar provides assistance to programmers in each of these scenarios. When multiple programmers work on a task simultaneously at one computer, the focus provided by the task context can make it easier for the multiple programmers to discuss the software and for the non-driving programmers to follow the actions of the driving programmer on the screen.

[4] Mylar also supports individual tasks known only to the programmer. In this paper, we will note significant features that we do not have room to discuss; for more details on any of these features and others, see the Mylar website (www.eclipse.org/mylar).

[5] www.bugzilla.org, verified 15/09/06



Retrieve context

Loads context from repository task into the workspace

Task: 155135: add support for changing ticket status in Trac rich editor

Repository: https://bugs.eclipse.org/bugs

Select context below:

Date	Author	Comment
2006-08-24 21:28	mik.kersten@eclipse.org	mylar/context/zip
2006-08-24 20:30	steffenp@gmx.de	mylar/context/zip

Figure 3 Context Sharing Initiated from the Task List

Mylar provides assistance for the other two scenarios by facilitating the sharing of task contexts. We have experimented with two means of sharing such contexts. First, most commonly, we attach the context used in solving a problem to the report describing the problem in a shared issue repository. Second, a context for a task can be exchanged in email. As Mylar is an open-source project, we prefer the first approach to provide transparency in the development process.

A programmer who wishes to work on a task left off by another programmer or who wishes to see how an issue was resolved can import the context for that task into their workspace (Figure 3). Similar to how one programmer can switch between tasks, a programmer importing a task context can switch to that context thereby accessing just that subset of the software system the original programmer had considered in completing the task. When contexts for tasks are stored in a shared repository, such as the Bugzilla repository used for the Mylar development itself, the repository becomes a richer source of knowledge about how to complete problems. Currently, the Mylar repository has contexts attached for 253 tasks (as of September 14, 2006). These shared contexts have made it much easier for Mylar's developers to work on reopened bug reports and to delegate partially completed tasks. The Mylar project also has a policy of attaching task contexts with submitted patches. This policy has made it much easier to apply dozens of contributed patches each month.

Mylar also helps focus communication between multiple programmers by providing a rich, focused interaction with a shared issue repository. In particular, Mylar supports the use of queries to watch for changes in particular categories of issues. For example, a query may be used to watch all updates made to an issue by a particular colleague. When the colleague adds a comment to an issue, the programmer will see the issue appear under the query in the Mylar Task List and will see an incoming arrow to represent changes have been made in the repository to the issue. When the programmer opens the issue, the view of the issue reflects the latest changes; for instance, except for the latest unseen comments, conversations are folded.

3. DISCUSSION

Another way to add more WYSIWYN support to IDEs is to provide recommendations. Mylar provides an experimental form of recommendation called Context Search, which for particularly interesting resources automatically runs and displays reference-based searches. For example, the Context Search may display the callers of a particular method that has a high-degree of interest. By knowing the context of a task, in a similar way, when a programmer begins work on a new problem report, it is possible to recommend previous problems completed in the past and to provide rich support in suggesting which parts of the system may be relevant to solving the problem [5]. These recommendations provide focus when they are sufficiently accurate; inaccurate recommendations would reduce the focus of the programmer.

At times, it would also be useful for collaboration to know which parts of the system are concurrently being worked on by other team members. One way of presenting this information is through decorations to resources in the IDE [1]. This decoration can be overwhelming when applied to all resources in the system. By knowing which resources programmers are working on and what they are doing with those resources, it may be possible to use task context to scope the collaboration information presented to provide more focus. For example, dynamically determining who the current team is working on a task and focusing collaboration affordances in the UI around that team.

4. SUMMARY

Most programmers' work is structured by the tasks that they perform. Mylar uses information gathered about how programmers work on tasks to focus the display of system information to the programmer and to focus the interaction of the programmer with the development environment. In this position paper, we have provided a brief overview of some of the facilities provided by Mylar to focus individual and team programming efforts. A full description of Mylar's features can be found at the project's website. Mylar ensures that what the programmer needs is what the programmer gets.

5. ACKNOWLEDGEMENTS

This work has been funded by NSERC and IBM. We are grateful to the many users of Mylar that have greatly enhanced its functionality.

6. REFERENCES

[1] Cheng, L., Hupfer, Susanne, Ross, Steven and Patterson, John. Jazzing up Eclipse with Collaborative Tools. *Proc. of the 2003 OOPSLA Workshop on Eclipse Technology Exchange*, pp. 45-49.

[2] Gonzales, V.M. and Mark G. Constant, constant, multi-tasking craziness: managing multiple working spheres. *Proc. of the Conf. on Human Factors in Computing Systems*, 2004, pp. 113-120.

[3] Kersten, M. and Murphy, G.C. Mylar: a degree-of-interest model for IDEs. *Proc. of the Conf. on Aspect-oriented Software Development*, 2005, pp. 159-168.

[4] Kersten, M. and Murphy, G.C. Using task context to improve programmer productivity. To appear, *Proc. of Conf. on Foundations of Software Engineering*, 2006.

[5] Murphy, G.C., Kersten, M., Robillard, M.P. and Čubranić. D. The emergent structure of development tasks. *Proc. of European Conf. on Object-oriented Programming*, 2005, pp. 33-48.

[6] Robillard, M.P., Murphy, G.C., and Coelho, W. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, Vol 30, No 12, 2004, pp. 889-903.

Visualizing Roles and Design Interactions in an Open Source Software Community

Flore Barcellini, Françoise Détienne
INRIA –CNAM Eiffel team
Domaine de Voluceau BP105
78153 Le Chesnay Cedex France
003313963{5255,5522}
{Flore.Barcellini, Francoise.Detienne}@inria.fr

Jean-Marie Burkhardt
Université Paris 5 ECI
45 rue des Saint-Pères
75270 Paris France
0033142862135
Jean-Marie.Burkhardt@univ-paris5.fr

Warren Sack
University of California Santa Cruz
Film and Digital Media Dept
1156 High Street
Santa Cruz, CA 95064, USA
0011831459-3204
wsack@ucsc.edu

ABSTRACT

We propose a means to visualize design-related, online discussions based on an analysis of the quotations shared between messages. We present an analysis of online discussions in an Open Source Software (OSS) design community. The objective of this research in cognitive ergonomics is to understand and to model the dynamics of social interactions that take place in OSS design mailing lists. We hope this research also informs the architecture of new tools for supporting organisational memory archives and the recording of design rationales. We show how an analysis of the quotation relations between messages can be used to locate design-relevant data in discussion archives and to retrace the thematic coherence of online discussions. Our analysis also reveals how the social structure of a design project influences the design process. Implications for the architecture of design visualization and design rationale tools for OSS development are outlined.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**] Electronic mail, H.5.3 [**Group and Organization Interfaces**] Asynchronous interaction, Theory and models.

General Terms

Design, Human Factors

Keywords

Distributed asynchronous design, quoting practices, Open Source Software projects

1. INTRODUCTION

In this communication, we propose several new ways to visualize online interactions between participants in Open Source Software (OSS) design-oriented online discussions. Our proposals are based on our research in cognitive ergonomics investigating both the dynamics of the OSS design process and some methodological principles to study activities in distant and asynchronous, mediated, design situations ([1], [2], [3]).

OSS design is a particular case of asynchronous, distributed, collaborative design. As analysed previously by Sack et al. [2], the OSS design activity occurs in three activity spaces: the discussion space, the documentation space and the implementation space. A large part of the OSS design process takes place in the discussion space and is archived in the documentation space. These traces of the design activity are thus important resources for users and developers of OSS. Considering the large quantity of data generated and archived, proposing methods and tools to extract relevant data for organizational memory [4] is a crucial issue for OSS researchers, OSS community building, and the efficacy of social interaction between participants in OSS projects, especially new comers in a project community [5].

This research is focused on a major OSS project devoted to the development of a programming language called Python. The designers of Python engage in a specific design process called Python Enhancement Proposals (PEPs). PEPs are the main means for proposing new features, for collecting community input on an issue, and for documenting chosen design decisions. Our message corpus was drawn from the python-dev mailing list that hosts PEP related discussions pertinent to design process.

Our approach is based on quotation as a relevant link between messages to reconstruct the thematic coherence and to locate design relevant data in online discussion archives. Until now the dominant model used to represent conversation, the threading model, has been based on reply-to links between messages. We have shown that quotation-based representations are more relevant than threading-based representations to reconstruct thematic coherence of design-oriented online discussions [3]. The quotation-based methodology developed in this study is also a powerful tool for studying online discussions and for highlighting the social interactions between participants during the design process; e.g., the roles played by project participants; the differences of influence between participants; and, the sequences of activities enacted during the design and implementation processes ([1], [3]).

Our research strategy is based on two complementary approaches: (i) analysis "by hand"; and, (ii) "automated" analysis of the online discussion corpora (i.e., the messages exchanged by the designers). The analysis “by hand” has been conducted to preliminarily test the validity of the quotation model to reconstruct the thematic coherence of design-oriented discussions and to analyse the design process. Based on these results, we automate parts of the structure and content processing. Currently under development is software to automatically identify quotation links between messages. We also hope to construct software to automatically analyse themes of discussion (cf., [6]).

In this communication we present the “by hand” analysis, discuss the validity of the quotation model for online discussions, and outline potential implications for architecture of future design tools.

2. QUOTING-BASED VISUALIZATION OF ONLINE DISCUSSIONS

2.1 Thematic coherence in online discussions

A large part of OSS design takes place in a discussion space where messages are exchanged between participants. A central aspect of thematic coherence concerns how any given message connects to previous messages. In face-to-face conversation, coherence relations are based on connections between conversational “turns” in a dialogue. For example, a question can constitute one turn and an answer another; a question-answer pair constitutes a coherent link within a conversation. Coherence in face-to-face conversation can be seen as actively constructed by participants across turn-taking. In contrast to face-to-face situations, in online conversations a message can be separated both in time and place from the message it responds to. Processes of turn-taking and topic (theme) maintenance are subject to disruption and breakdown [7]. In online discussions, messages are posted by geographically-distributed participants in an asynchronous manner. When examined chronologically – i.e., in the order received by the system -- there are indeed disrupted turn adjacencies: turns that are intended as responses or follow-ups to previous turns, do not occur temporally adjacent to initiating turns [7]. This sort of disruption is a violation of sequential coherence one normally expects in face-to-face conversation (pragmatic principles of adjacency and relevance). This can create potential confusion that users seek to minimize by adopting compensatory strategies for conversational linking. Quotation is one such strategy.

2.2 An alternative approach to the threading-based representation of online discussions

Quotation is a widely used technique in emails dialogues and forum discussions [8]. Quotation creates the illusion of adjacency: it incorporates portions of two turns within a single message. It maintains context (i.e., portions of previous messages) and so can be used to retrace the history of a conversation [7].

As far as we know, there have been only two attempts to develop tools to automatically identify quotations and to represent online conversations based on quotation links between messages: CONVERSATION MAP [9] and a prototype inspired by CONVERSATION MAP called ZEST [10]. The thread-based approach is still the main basis of tools for organizing online discussions. Mixed models of visualization combine this approach with the sequential model (e.g. [11]). These representations are useful for analyzing interactional roles in conversations. They provide a picture of the centrality (versus periphery) of participants in the community of posters (e.g., [12]). Central participants may be considered as those who tend to get more replies to their posted messages (see [13]). However their relevance for identifying and visualizing the thematic coherence of online discussions may be questioned on the basis of computer-mediated communication studies presented above.

In a previous paper we argued that our quotation-based representation appears to be more relevant than a threading model for reconstructing the thematic coherence of design-related online discussions. Our analysis of quotation practices allows us to compare a representation of PEP-related online discussions (Figure 1) with a representation based on threading or “reply-to” links between messages (Figure 2). In the figures, the circles or squares represent email messages (labeled with an arbitrary number). Arrows joining the circles symbolize either a “is-a-reply-to” or a “is-quoted-by” link between two messages. The circles or squares are displayed differently to represent the theme (i.e., the different design problems) addressed by the messages. Using the reply-to links to partition the messages (Figure 1), it appears to be the case that the conversation is fragmented into several threads. This analysis by threads also corresponds to the way in which the discussion is archived on the web (at the URLs cited above). The quotation-based visualization (Figure 2) reveals a distinctly different organization of the messages. The thematic coherence of the discussion, especially regarding Theme 1 (T1), is better represented by the quotation-based links (Figure 2) than by the reply-to links (Figure 1). In this quotation-based representation of discussions (Figure 2) all of the messages are connected together. Closer examination of the message contents reveals that the messages that are unlinked in Figure 1 are pivotal to the overall discussion. The longevity of discussion themes is dependent on its relevance to the PEP. In general, in design discussions, discussion themes do not dissipate over time. This is one way in which design discussions differ from open online discussions where discussion themes general do disappear over the course of an online exchange [7].

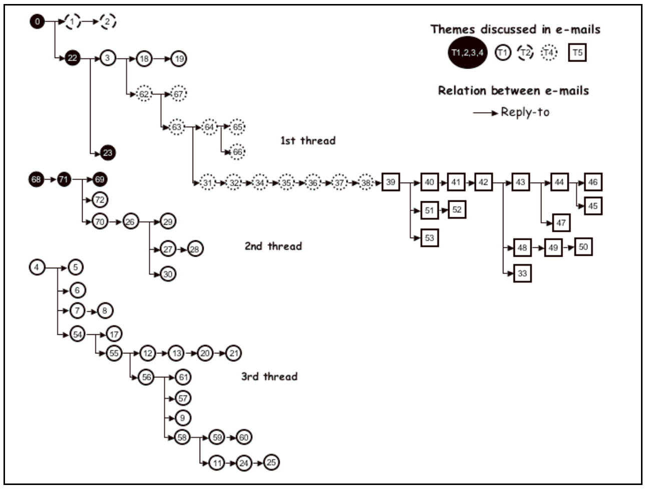


Figure 1: Threading based representation of the links between messages PEP 279

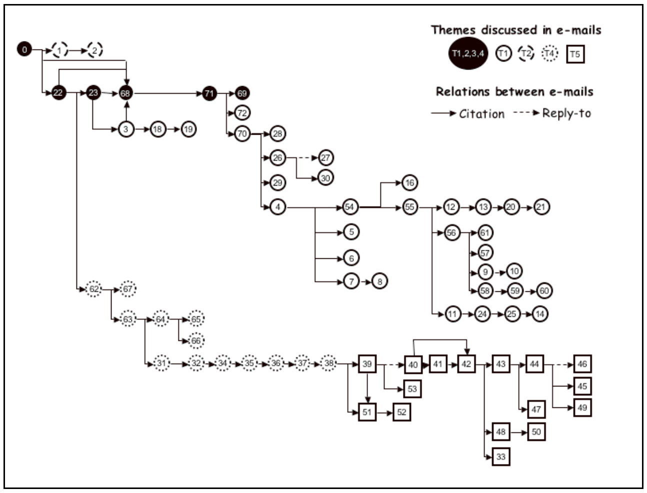


Figure 2: Quotation-based representation of the links between messages PEP 279

3. SOCIAL INTERACTIONS, TEMPORALITY AND ACTIVITIES IN DESIGN ORIENTED ONLINE DISCUSSIONS

We used this quotation-based visualization to investigate the social interactions, the temporality, and the design-related activities of the online, design discussions.

The social interactions are represented using a quotation-based visualization of discussions in which participants' roles are highlighted (Figure 3). Messages are labelled with the project roles of the posters. Three important, project roles -- related to the PEP process -- in the Python community have been identified [14]: (1) the project leader; (2) the administrators, whose role is to maintain the code base, the documentation, and the PEP process; and, (3) the developers. To distinguish levels of participation (high or low participation, HP or LP) in the online discussion, we have divided the population into two groups according to the median number of messages posted. Figure 3 shows that the patterns of quotation -- sequential versus branch structure -- tend to correspond with the social position of the poster in the Python project: (1) a branching structure (when multiple messages quote from a single message) is generally initiated by a message posted by either the project leader or the PEP's champion (the one who proposed the new idea and wrote the PEP); (2) High-participant Administrators are usually the ones to post messages that close a line of discussion; (3) sequential structures tend to alternate between messages posted by administrators and messages posted by developers. Thus a participant's assigned role in the project organization affects who responses to the participant in the online discussion and, therefore, influences the unfolding of the design process within the discussion space.

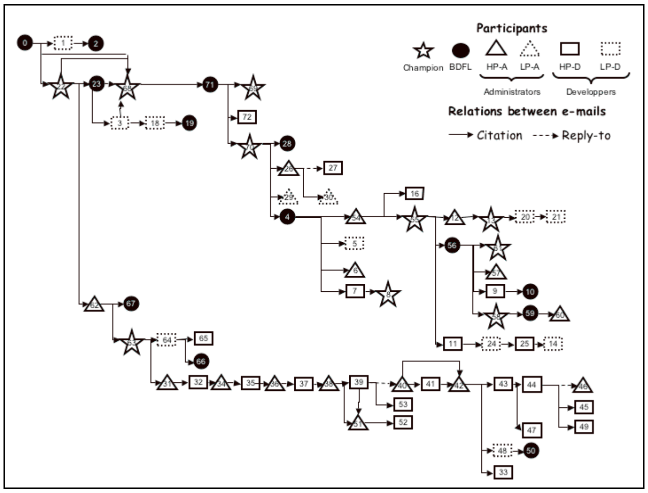


Figure 3: Status and position in the discussion PEP 279

We have examined the temporality of the discussions by ordering the messages chronologically (according to the time their were received by the server). We note that several clusters of messages can be discerned in which all of the messages of a cluster arrive within an hour of one another. These "quasi-synchronous" exchanges are usually focussed on a single theme of discussion. Examining the sequencing of these quasi-synchronous exchanges reveals a specific ordering of design steps taken by the community.

Finally, using a methodology of content analysis [15], we have analyzed the sequences of quotations and comments of the discussion to understand the "work flow" or design-related activities of the community's design process [1].

4. IMPLICATIONS FOR TOOLS AND FURTHER WORKS

Our results may have some implications for the design of tools for CSCW, especially for notification tools [18]; design-rationale tools [16]; and, organizational memory tools [4]; and for tools supporting large scale research on OSS development.

Our representation of online discussions highlights quotations between messages rather than simply their "reply-to" links. The quotation-based representation provides a means to visualize and analyze both the thematic coherence and the social dynamics of online discussions. Until now, most tools for organizing and visualizing online discussions have been threading-based. We believe that a quotation-based representation could provide a promising new approach for the design of CSCW tools.

Our quotation-based representation might be enriched by several user-adaptable functionalities inspired by our three levels of analysis (social, temporal and design-related). By automating some of our analysis methods, one might build a tool to display, for instance, quoted messages, messages that have been quoted multiple times, or beginning, branching messages that might be pivotal; e.g., those posted by the project leader, the champion, or the administrators; messages containing multiple quotations; messages that have been deeply quoted in discussions; etc. A tool designed to perform our temporal analysis might display only messages that have been quoted in the past day; or messages that have led to major synchronous activity. All these characteristics can be find automatically in discussions. We believe that tools based on our analysis methods could be helpful for developers, new comers or lurkers to find relevant information and facilitate organizational memory.

We imagine a design rationale tool that could display sequences of quotations and comments that are linked to argumentation. Such a tool might make explicit the design rationale, i.e., the reasoning behind the design of a software artifact. By making their rationales explicit, designers will be able to keep track of past decisions and communicate these rationales to others outside the design team. Two approaches could be used to develop this kind of tool. The first approach would require users to tag messages to categorize the content and design rationale expressed in the messages [17]. The main shortcoming of this approach is that it creates an added task for the users. The second approach would be to construct an automatic discourse tagger to analyze automatically the themes of discussion and patterns of argumentation, an admittedly difficult task akin to rhetorical structure parsing [6].

5. REFERENCES

[1] Barcellini, F, Détienne, F., Burkhardt, J.M., et Sack, W. (2005). Thematic coherence and quotation practices in OSS design-oriented online discussions. In K. Schmidt, M. Pendergast, M. Ackerman, et G. Mark (Eds.) *Proceedings of the 2005 International ACM SIGGROUP* (pp 177-186). New York, USA: ACM Press.

[2] Sack, W, Détienne F, Burkhardt, J.M., Barcellini F, Ducheneaut, N, and Mahendran D. (in press). A Methodological Framework for Socio-Cognitive Analyses of Collaborative Design of Open Source Software. *International Journal of Computer Supported Collaborative Work, special issue on Distributed Collective Practices.*

[3] Barcellini, F, Détienne, F., Burkhardt, JM., Sack, W. (2005). A study of online discussions in an Open-Source community : reconstructing thematic coherence and argumentation from quotation practices. In Van Den Besselaar, P., De Michelis, G., Preece, J., Simone, C. (Eds) *Communities and Technologies2005 (pp 301-320),* Dortmund, The Netherlands, Springer.

[4] Sauvagnac C., and Falzon, P. (2003). Organizational memory: the product of a reflexive activity. *The International Journal of Cognitive Technology, 8(1*), 54-60.

[5] Ripoche, G., and Sansonnet, J-P. (in press). Experiences in automating the analysis of linguistic interactions for the study of distributed collective. *Journal of Computer Supported Collaborative Work, special issue in Distributed Collective Practices.*

[6] Marcu, Daniel (1997) *The Rhetorical Parsing, Summarization, and Generation of Natural Language Texts*. Ph.D. Dissertation, Department of Computer Science, University of Toronto, Toronto, Canada, December 1997

[7] Herring, S. (1999) Interactional Coherence in CMC. In *Proceedings of the 32nd Hawaii Conference on system sciences,* 1999.

[8] Eklundh, K.S, and Macdonald, C. (1994) The use of quoting to preserve context in electronic mail dialogues. *IEEE Transactions on Professional communication*, vol.37, n°4, (pp197-202).

[9] Sack, W. (2000) Conversation Map: A content-based Usenet newsgroup browser. In Proc *IUI 2000,* ACM Press, 233-240.

[10] Yee, K-P. (2002) Zest: discussion mapping for mailing lists. *CSCW 2002* (demo).

[11] Venolia, G., and Neustaedter, C. (2003) Understanding sequence and reply relationships within email conversations : a mixed-model visualization. In *Proceedings of CHI 2003,* April 5-10, Florida, USA.

[12] de Souza, C. R. B., Froehlich, J., & Dourish, P. (2005) Seeking the Source: Software Source Code as a Social and Technical Artifact. *ACM International Conference on Supporting Group Work* (GROUP 2005), pp: 197-206.

[13] Viégas, Fernanda B., Marc Smith. (2004). Newsgroup Crowds and AuthorLines: Visualizing the Activity of Individuals in Conversational Cyberspaces. In *Proceedings of the 37th Hawaii Conference on system sciences.*

[14] Mahendran, D. (2002) *Serpents and Primitives: An ethnographic excursion into an Open Source community.* Master's Thesis, School of Information Management and Systems, UC Berkeley, May 2002.

[15] d'Astous, P., Détienne, F., Visser, W., and Robillard, P. N. (2004). Changing our view on design evaluation meetings methodology: a study of software technical evaluation meetings. *Design Studies, 25*, 625-655.

[16] Moran, T. P., Caroll, J. M. (1996) *Design rationale: concepts, techniques, and use*. Mahwah, NJ, USA: Laurence Erlbaum Publisher.

[17] Kirschner, P.A, Buckingham Shum, S.J., and Carr C. S. (2003) *Visualizing Argumentation: Software Tools for Collaborative and Educational Sense-Making*. Springer-Verlag: London.

[18] Carroll, J., M., Neale, D., C., Isenhour, Philip; L., Rosson, M.B., McCrickard, D.S. (2003). Notification and awareness: synchronizing task-oriented collaborative activity. *IJHCS, 58*,605-632

Supporting harmonious cooperation in global software development projects

Anders Sigfridsson
Interaction Design Center
Engineering Research Building
University of Limerick, Ireland
+353 86 23 33 625
anders.sigfridsson@ul.ie

Henrik Dahlgren
BassetLabs
Allén 6C P.O. Box 1156
SE-172 23 Sundbyberg, Sweden
+46 70 420 7657
henrik.dahlgren@bassetlabs.com

ABSTRACT

In this paper we present the results from a field study at one site that is part of a large, multinational organization. The site is devoted to software development in cooperation with other, geographically distant sites in the same organization. Our focus is the cooperation between the different sites on a developer level and what prerequisites and tools are essential for this cooperation's potential of being harmonious. The purpose is to evaluate what tools, methods, and strategies are most promising to apply for managers assigned to organize distributed software development projects. The results indicate that the necessary support needed for harmonious cooperation includes telephone- and video conference tools and text-based communication tools such as e-mail, instant messaging, and chats. Supplementary face-to-face meetings such as kick-off meetings, recurring co-located meetings, and inviting experts from remote sites are also needed. As are complementary tools and strategies such as file sharing and version tracking tools, iterative development methods, project status tracking and communication of progress, and common spoken language in all sites. But in the end it is the individuals themselves, with their social competence, preferences, and relationships, who determine whether there will be cooperation or not. They have to actively choose to use the available supportive tools and assistance. Our conclusion is that harmonious cooperation depends on individual developers being conscious of the known challenges of cooperating across distance and actively adapting their personal work practices to that knowledge.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management – *software development, software process.*

General Terms

Management, Human Factors.

Keywords

Global software development, cooperation, support, tools, strategies, field study.

CSCW'06, November 4-10, 2006, Banff, Alberta, Canada.
Copyright 2006 ACM 1-58113-000-0/00/0004…$5.00.

1. INTRODUCTION

The process of developing a large software system is a cumbersome and complex one, in view of the fact that it includes the combined work of many people of different professions - from system designers and programmers to domain experts and managers - in a field that is uncertain and intricate. Certain characteristics of software development, such as large scale, uncertainty, and complex interdependencies, make control and management crucial if a workforce is to be engaged in an efficient way [6]. The trend today is that software development to an increasing extent is being distributed among geographically dispersed sites, evolving into a phenomenon that is often referred to as "global software development". When two or more remote sites are to cooperate across a distance in the development of software the strain on the cooperation and the need for coordination is even more significant [5, 7].

According to Lanubile et. al. [7], the three main challenges in global software development are the lack of informal communication, the cultural differences between distant sites, and the difficulty of building trust among remote developers. In current practice, these and many other issues are often visible, and together they amplify the difficulties of successfully organizing distributed software development projects. There is also a large plethora of tools, strategies, and methods that are often used in global software development projects in an effort to overcome these issues. For example, communication tools such as e-mail, instant messaging, and various chat tools are often mentioned, as are kick-off meetings and initial cultural training, and managerial efforts like de-centralization of decision-making and standardization of development environments [4, 7, 10].

The aim with this study was to examine the cooperation in actual, day-to-day practice for individual developers who work in distributed projects and to get an understanding of how the distribution – i.e. issues such as those mentioned above - affects this aspect, as well as what can be done to dampen the effects by deploying diverse strategies, methods, and tools. We have investigated what prerequisites and tools are essential for this cooperation's potential of being harmonious through a field study at one software development office which cooperates with other, distant sites on a daily basis. It is cooperation from the developer's point of view and what they perceive to be harmonious cooperation that we have looked at, i.e. when we say "harmonious" cooperation we simply refer to situations where the developers themselves do not view the necessary interactions with remote sites as disturbing for their work. The objective of the study was that the insights and conclusions regarding what makes cooperation harmonious and what tools, methods, and strategies are most promising to support this could be used as an approach to organizing distributed software development projects.

2. THE STUDY

The study that is presented in this paper was conducted at a local software development office in Umeå, Sweden, housing around 17 developers plus managers. The site is part of a large, multinational organization mainly devoted to investment banking. Part of the activities in the organization is development, adaptation, and analysis of electronic access to markets and trading desks around the world. It is these activities that the Umeå site is participating in, which means that they must interact daily with other sites within the organization around the world – primarily in New York, London, and Tokyo – on both development and management levels.

For this study, we interviewed a small number of experienced developers, both individually and in one group interview. The interviewees were carefully chosen in collaboration with the Umeå site manager to reflect the different types of work that is conducted at the site. The reason that we choose a small number of experienced developers was that we wanted to avoid getting caught up with issues that come from inexperience and also to steer clear of project specific issues and be able to get a more general idea of the work. The interactions with the participants were mainly directed at gaining an understanding of their daily work and their perceptions of the cooperation with other sites.

A comprehensive literature study in the field of global software development was performed before the field study started. The result from this were lists of known issues of cooperating in distributed software development and common supportive tools methods, and strategies that are available to overcome, counter, or avoid those issues. At the site, the group interview was conducted first, in an attempt to get an initial impression of the developers work practice and what their view of issues and supportive tools, methods, and strategies was. The lists deducted from the literature study were not revealed during this event. Instead they were re-designed afterwards to incorporate the insights that surfaced during the group interview, so that, for example, some issues were added while others were removed. During the individual interviews we introduced our findings from the literature study and the group interview. This then served as a ground for further discussions with each of the individuals, which allowed us to develop our understanding even more in detail.

3. VITAL SUPPORTIVE TOOLS, METHODS, AND STRATEGIES

3.1 Communication and social possibilities

The individual issues that were deemed the most significant all mainly affect communication and social opportunities and abilities, albeit representing different approaches to these aspects. There are several supportive tools, methods, and strategies aimed at these aspects that are seen as effective and thus considered vital. Above all, the techniques in the category communication tools are rated as effective and do aim to provide communication opportunities, both formal and informal. Therefore, telephone- and videoconference possibilities as well as tools that support text-based informal communication – e.g. instant messaging, e-mail, and chats – are the most vital tools indicated in the study. These also provide the possibilities of building social relationships with participants in other sites. Taken as a whole, providing good communication channels and hence the possibilities of mimicking the casual interactions that face-to-face cooperation offers and of building a coherent social network within and between projects, has been expressed as the most vital supportive strategy in our study.

Although none of the members in the face-to-face opportunities category are among the ones considered most effective, the things that were said about them during the interviews combined with the fact that lack of formal face-to-face meetings is one of the most significant issues makes them essential as well. Taking into account what has been said during the interviews, it is not just the coordination potential that kick-off meetings, co-located meetings, or inviting experts from remote sites to participate in local work offers that makes them vital – the social potential is a major contributor to their importance.

3.2 Coordination and technical support

File sharing and version tracking tools, project status tracking and communication of progress, and iterative development methods were believed to be effective tools and strategies to support cooperation. They all mainly target coordination and technical aspects of the work in distributed projects. However, the findings regarding issues do not highlight challenges in these areas as particularly significant. Especially technical issues like version conflicts and incompatibility problems are not deemed influential. But it also has to be taken into consideration that all of these tools and strategies are said to be very frequent in projects.

These and a few others – notably common spoken language in all sites - are considered such an integrated part of the work environment in distributed projects that they are taken for granted by the developers. The communication tools belong to this group too, and are also seen as very important. However, the fact that these other tools and strategies are not explicitly considered vital does not mean they are dispensable – they are such an integrated part of the work environment that they influence what issues are considered significant simply because no one pictured how things would work without them. Although the respondents do not as clearly request those aids as the communication tools or face-to-face possibilities mentioned above, they are nonetheless important for the work simply because they enable the daily cooperation activities.

During the interviews, one of the respondents pointed out that it is the combination of communication possibilities like instant messaging, e-mail, and telephone conferences that are vital, not the individual techniques by themselves. This point is a general one and must be taken into account when considering what tools and strategies are vital and not – focusing on providing the most essential ones do not mean discarding all the other, but rather to try reaching the emergent function of a combination. As a consequence of this insight, the study shows that communication tools and face-to-face opportunities are considered most vital, while cooperative work tools such as file sharing and version tracking tools, methodological efforts like iterative development, efforts to enhance project awareness like project status tracking and communication of progress, and strategic efforts like ensuring common spoken language in all sites are obligatory present as a transparent supportive scaffolding.

4. HARMONIOUS COOPERATION

4.1 Experience and consciousness

During the interviews, the respondents have repeatedly stated how people learn to adapt their work practice to certain issues like cultural differences and cooperating without a great deal of face-to-face meetings. This has been exposed throughout the study as well. For example, as to why cultural differences were not considered a very significant issue people gave the reason that they learned to adapt to the differences, and the same reason is given for different time zones not being very influential.

Time, it seems, puts everything in perspective – individual participants learn to adjust their efforts to overcome challenges like cultural differences and counter the lack of informal communication and formal face-to-face meetings; as projects evolve the coordinators learn to navigate and structure them around challenges like different time zones and achieving appropriate levels of centralized or de-centralized decision making; the organization as a whole grows more mature and learns how to manage the interdependencies, knowledge, and responsibilities so that known issues like unclear task distribution and diverse market influence on different sites are avoided or at least minimized.

The key in this chain of events is gaining and utilizing experience on individual, project, and organizational levels. Gaining experience is essentially becoming more and more aware of ones surroundings and consciously acting in response to that knowledge. When individual developers become more experienced in working in distributed software development projects they begin to work more and more consciously with respect to what is needed for good cooperation. We use the word "conscious" because what we mean is that they both become aware of issues and how to overcome them, and begin to take personal responsibility to adapting their work practice to this awareness.

4.2 Achieving harmonious cooperation in global software development

Working together harmoniously implies being aware of the interdependencies with other activities and acting with respect to that [8, 9]. In global software development, working together harmoniously seems to include not only being conscious of and acting in response to the interdependencies between remote sites, but also being conscious of and acting in response to challenges like cultural differences, lack of formal face-to-face meetings, and impeded informal communication.

The necessary elements for achieving harmonious cooperation in distributed projects therefore include not only providing the vital communication tools together with supplementary face-to-face opportunities and the obligatory environmental interior – i.e. version tracking tools, common spoken language at all sites, iterative development methods, etcetera – but also building and retaining attentiveness of challenges and interdependencies on a individual level. On a strategic and tactical level it is a matter of providing the specific aids identified as vital and obligatory as well as coordinating the projects according to the interdependencies between sites and projects. On an operational level it is about adapting the daily work practices to knowledge about issues of cooperating across a distance. This means that, in the end, it is the individual developers themselves who determine whether or not there will be cooperation and they must actively choose to make use of the available support for this.

4.3 Applying a corporate culture

The natural reaction for many to the points we make above is that training is the solution. To a certain extent we agree with this; it is certainly possible to build some awareness of known issues and how to avoid them by educating people formally. However, we believe that the results of this approach are very limited. Based on what the participants of this study have expressed, formal training in, for example, cultural differences, is not something that is vital in achieving harmonious cooperation. Instead, the general opinion has explicitly been that you need to experience the actual situations to learn how to deal with them and that it is only then that you can build the necessary consciousness. We can also find theoretical support for this view. For example, Brown et al. [2] metaphorically refers to knowledge as a tool in the sense that one can only fully understand it through use and that using it means adopting the belief system of the culture in which it is spawned and used. From this point of view, learning is about enculturation of the learners into authentic practices through activity and social interaction. In other words, learning is best done through actual experience and apprenticeships.

The organization that the Umea site is a part of has a strong corporate culture and it has been stressed during the interviews how important this is for good cooperation because it dampens the impact of local differences and other issues discussed in this paper. For example, the organization is actively spreading its corporate culture to all offices within the company by transferring experienced employees to new offices in an effort to make good use of the collective experience and to help a new office to work more consciously. Another aspect of the particular setting of the Umea site also has to be noted. There is a relatively high degree of stability regarding the sites and people the developers in Umea have to cooperate with. The situation is such that after working for a while in the organization the developers are able to form personal relationships with people in other sites which continue after the projects come to an end.

Our opinion about this situation is that what we in fact are seeing is in essence the spreading of experience and consciousness throughout the organization. Since the spreading of experience is a by-product of building a corporate culture in the way we have seen in this study, this creates the necessary experience, while the stability gives individuals the chance to actually adapt to what they experience. Instead of aiming solely for training as a way to achieve what we have identified as vital for harmonious cooperation, we suggest that strategies like those mentioned above – i.e. pushing a strong corporate culture and creating as much stability as possible in and between projects – are good examples of ways to enable developers to work more consciously.

5. FURTHER WORK

Some findings that are similar to those of this study can be found in the literature focused on the coordination and collaboration mechanisms of open source software communities. For example, Gutwin et al. [3] demonstrates how simple text based tools such as email lists and chats are surprisingly effective coordination mechanisms in open source projects. It is believed that the media themselves have several characteristics that make them appropriate for this – e.g. Gutwin et al. highlights the value of public access and overhearing in implicitly building general awareness and the idea of an active mailing list as a proxy for identifying the appropriate people. However, it is only in combination with the fact that the participants both voluntarily and frequently use and rely on these tools that this is made possible and successful. Further research into the open source communities and the ways they collaborate might provide useful insights regarding cooperation in software development which can be transferred to corporate settings. We consider this area worthy attention to continue the work that has been initiated here.

Also, there are some interesting theoretical concepts that might provide useful extensions of the approach we suggest here. One possible theoretical pillar might be found in Bannon [1]. He argues that a user is not to be seen as a passive 'factor', but as an active 'actor', a person who is part of a context and work community and has motives and preferences, as well as expertise and skill. One could make a similar argument here: we should not view the software developers as passive and dim-witted by thinking that so long as they have advanced tools to help them in their social interactions they will be able to cooperate and will do so harmoniously. Instead, the role of the tools is to readily provide a framework for them to use – the crucial thing is consciousness and, above all, motivation in the development team so that they will use the framework and willingly cooperate.

6. CONCLUSIONS

The necessary support needed for harmonious cooperation includes telephone- and video conference tools and text-based communication tools such as e-mail, instant messaging, and chats. Supplementary face-to-face opportunities such as kick-off meetings, recurring co-located meetings, and inviting experts from remote sites are also needed. As are complementary tools and strategies such as file sharing and version tracking tools, iterative development methods, project status tracking and communication of progress, and common spoken language in all sites.

The conclusion of this study is that harmonious cooperation depends on individual developers being conscious of the known challenges of cooperating with someone across distance and actively adapting to that knowledge. In the end, it is the individual developers themselves who determine whether or not there will be cooperation and they are the ones who actively choose to make use of the available support for this.

Our study has shown that experience is an essential part in this process since it leads to consciousness and adjustment of personal work practice. We have also seen that a stable environment during and between projects and a strong corporate culture are good examples of how this kind of experience can be spread throughout the organization. Instead of applying the tools and strategies identified as vital for supporting harmonious cooperation and to then rely solely on training to achieve the necessary consciousness, we suggest that strategies like pushing a strong corporate culture and creating as much stability as possible in and between projects are a good way to enable developers to work more consciously.

7. ACKNOWLEDGEMENTS

The authors would like to thank the participants of the study and the coordinators at the site who helped us in our work, as well as all the people who read drafts of this paper and gave us constructive critique and guidance. This applies especially to Mikael Wiberg at Umea University, Sweden, for his supervision of the study.

8. REFERENCES

[1] Bannon L. J. (1991) From Human Factors to Human Actors – The Role of Psychology and Human-Computer Interaction Studies in Systems Design. Chapter in Greenbaum J. & Kyng M. (Eds.) (1991) *Design at work: Cooperative design of computer systems*. Hillsdale: Lawrence Erlbaum Associates, pp. 25-44.

[2] Brown J. S., Collins A. & Duguid P. (1989) Situated Cognition and the Culture of Learning. *Education researcher*, vol. 18, no. 1, pp. 32-42.

[3] Gutwin C., Penner R. & Schneider K. (2004) Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pp. 72 – 81, 2004.

[4] Hargreaves E., Damian D., Lanubile F. & Chisan J. (2004) Global Software Development: Building a Research Community. In *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, September 2004, pp. 1-5.

[5] Herbsleb J. D., Mockus A., Finholt T. A. & Grinter R. E. (2000) Distance, Dependencies, and Delay in a Global Collaboration. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, December 2000, pp. 319-328.

[6] Kraut R. E. & Streeter L. A. (1995) Coordination in Software Development. In *Communications of the ACM*, vol. 38, no. 3, March 1995, pp. 69-81.

[7] Lanubile F., Damian D. & Oppenheimer H. L. (2003) Global Software Development: Technical, Organizational, and Social Challenges. In *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, November 2003, pp. 1-4.

[8] Malone W. T. & Crowston K. (1990) What is Coordination Theory and How Can It Help Design Cooperative Work Systems? In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, October 1990, pp. 357-370.

[9] Malone W. T. & Crowston K. (1994) The Interdisciplinary Study of Coordination. In *ACM Computing Surveys*, vol. 26, no. 1, March 1994, pp. 87-119.

[10] Sarma A. (2005) *A Survey of Collaborative Tools in Software Development.* ISR technical Report # UCI-ISR-05-3. Institute for Software Research, University of California, Irvine, March 2005.

Supporting Software Development as Knowledge Community Evolution

Kumiyo Nakakoji[1,2] Yasuhiro Yamamoto[1] Yunwen Ye[2,3]

[1]RCAST, University of Tokyo
4-6-1 Komaba
Meguro, Tokyo, 153-8904, Japan

[2]SRA-KTL Inc.
3-12 Yotsuya
Shinjyuku, Tokyo, 160-0004, Japan

[3]Dept. of Computer Science
University of Colorado, Boulder
CB430 Boulder, CO. 80303

kumiyo@kid.rcast.u-tokyo.ac.jp yxy@kid.rcast.u-tokyo.ac.jp yunwen@cs.colorado.edu

ABSTRACT

We view software project as a knowledge ecology consisting of three interrelated elements: (1) artifacts, (2) individual developers, and (3) a community of developers. How developers relate with each other in the community affects how they share knowledge during the development and therefore impacts the overall quality of the software system that have to be built through continuous knowledge collaboration. This paper analyzes this social relation and its impacts on software development, and presents an approach to help developers make use of peer expertise by asking and helping other developers. It then describes the STeP_IN (Socio-Technical Platform for In situ Networking) framework to illustrate the approach.

Keywords

Knowledge collaboration, knowledge community, software development, reuse, community, socially aware communication, socio-technical approach

1. SOFTWARE DEVELOPMENT AS KNOWLEDGE COLLABORATION

The development of large-scale software systems is a social activity, carried out through the collaboration by a group of software developers. The social aspects of software development have been studied mostly in the context of how developers and users work together in designing systems [22], in the organizational context of a software project [17], or in distributed software development teams [8]. This position paper in contrast focuses on the knowledge collaboration of software developers: how developers can make use of peer expertise in collectively creating the software system.

Software development is essentially a knowledge construction process that needs knowledge in a variety of fields, which is constantly changing. For example, application domains are subject to rapid change; component libraries are continually updated; new features and functionalities continue to be introduced in programming tools and environments. Software development can therefore be viewed as a learning process and software developers have to constantly acquire new knowledge.

It may come as a surprise that software developers also need to learn about the system that they are developing. One may argue that since the software developer participates in the creation of the system, he/she should know the system inside out. However, because large scale software systems are created collaboratively by many developers, not all developers, if any, would have complete knowledge about the whole system. At the same time, With the increasingly widely accepted view of software systems as evolving entities, the percentage of incremental, continuous development tasks in software development has risen quickly. Such software systems need to be continuously developed with iterative processes to adapt to the ever-changing user requirements and execution environments. Coupled with the high turnover rate in software industry, many software developers find themselves working to make incremental changes to systems that have been partially developed, or even are operating on a daily base (such as those web-based systems) .

For software developers, software code is the ultimate knowledge resource about the system. During the development process, they intensively engage in recovering "implicit knowledge" embedded within the code [11]. Due to the essential invisibility of software code, however, the needs of creating documents that provide high level descriptions of the code and the design rationale have been recognized.

Code and documents, however, are often still not enough. Documents often do not exist or are not in sync with the code. Moreover, a culture exists in software development that prevents developers from sharing knowledge over the entire source code. As LaToza et al. observed, "implicit knowledge retention is made possible by a strong, yet often implicit, sense of code ownership, the practice of a developer or a team being responsible for fixing bugs and writing new features in a well defined section of code" [11]. Much of the knowledge about the code and the design decisions remain in the head of developers. This "symmetry of ignorance" [4] within a development team is neither a problem nor an accident; it is a matter of fact in software development,

Supporting knowledge collaboration among software developers thus becomes an important research topic in supporting software development. This paper first conceptualizes software project as a knowledge ecology that has intertwined and dynamically changing relationships among software artifacts (code and documents), software developers, and developer community, followed by the analysis of social factors in supporting knowledge collaboration in software development based on this conceptualization. Finally, the paper describes the STeP_IN (Socio-Technical Platform for In situ Networking) framework that supports knowledge collaboration in software development by taking into full consideration those identified social factors.

2. THREE ELEMENTS IN SOFTWARE DEVELOPMENT

We view software project as a knowledge ecology that consists of three interrelated elements: (1) artifacts, (2) individual developers, and (3) a community of developers (Figure 1). A group of developers engaging in software development can be viewed as a knowledge community, defined as a group of people who collaborate with one another for the construction of artifacts of lasting value [2]. In a knowledge community, people are bonded through the construction of common artifacts.

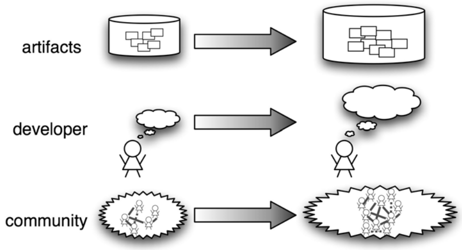


Figure 1: Software Project as a Knowledge Ecology Consisted of Three Interrelated Elements

The community element is essential when viewing software development as collective creative knowledge work. The roles of individual developers, both formally assigned ones and informally perceived ones, change over time during a project. The social relationships among the developers grow through their engagement in the project, affecting how they collaborate, communicate, and coordinate with one another, which results in different ways of sharing knowledge.

Because knowledge sharing is indispensable in software development, the quality of the resulting software depends not only on the skills and knowledge of individual developers, but also on the roles and social relationships among the developers. In other words, the quality of the software to be developed is determined not only by the sum of each developer's knowledge, but also on the social relationships of software developers that impacts the sharing of knowledge during the development process.

All three elements constantly evolve during the process of software development. Artifacts change over time throughout the development. Individual developers—or, more precisely, what individual developers know—grow by gaining experience through the engagement with artifacts and peer developers. The community of developers changes when new developers join, old developers leave, and both the assigned and perceived roles of members change.

Existing studies on understanding and supporting software evolution have primarily focused on the evolution of artifacts. More recent work has started to look at how individuals change through learning about the system. People learn by reading source code and documents, and they learn by asking peers questions. They also learn by solving new problems and experiencing unfamiliar situations. Their old knowledge is replaced with new knowledge and is restructured during the development process.

In contrast, not much has been studied on the aspects of the evolution of the developer community in the context of software development [15]. A community evolves through individual activities in software development that result in either the change of software artifacts or the individual growth of knowledge about the system. This paper views the evoluationary process of a community from the following three relationships (Figure 2).

(1) *The relationship of an individual with artifacts.* How one relates with artifacts is concerned with what knowledge, expertise, and experience the individual has on what artifacts. This information is useful in identifying a set of people who are likely to have expertise with a certain artifact.

(2) *The relationship of an individual with other developers.* How one relates with other individuals impacts social relationships among developers. This information helps a developer determine whom to ask for help about a certain artifact as well as decide whether and how to respond to a question being posed by an asker (Figure 3).

(3) *The relationship of an individual with the community as a whole.* How one relates to the community is concerned with that individual's role within the community: whether he/she is a peripheral member, a core member, or a member in between. This relationship helps a developer decide how much he/she should contribute to the community by gaining trust and social reputation within the community. One's role evolves within a community through legitimate peripheral participation [21]. By looking at how and what a developer's peers who are closer to the core of the community do within the community, the developer gradually acquires skills through learning, and develops his/her identity within the community.

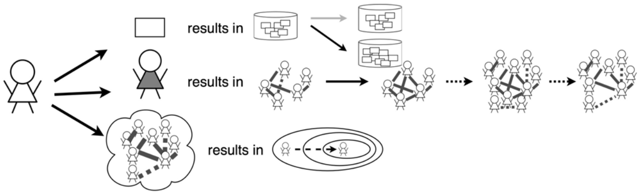


Figure 2: Three Aspects of the Community's Evolutionary Process

3. SOCIAL FACTORS IN SOFTWARE DEVELOPMENT

To support software project as a knowledge ecology that consists of the interrelationship among software artifacts, individual developers, and developer community, we have focused on the following aspect: how to help developers make use of peer expertise in development activities.

A number of researchers have already recognized the needs of using the expertise of other software developers. Berlin has found that expert developers are experts not only because they have more expertise but are able to use other experts more [1]. Several systems, notably Expertise Recommender [12] and Expertise Browser [13] that help software developers to find experts, have been proposed in the past years.

Finding experts, however, does not necessarily lead to the acquisition of their expertise [23]. As knowledge resources, experts are different from other resources that are things. "A thing is available at the bidding of the user—or could be—whereas a person formally becomes a skill resource only when he consents to do so, and he can also restrict time, place, and method as he chooses" [9].

Thus, when peers' expertise becomes critical resources for a programming task, simply knowing who has the expertise is not enough. The expertise seeker (i.e., asker) needs to establish a communication channel with the potential expertise providers (i.e., helpers) and asks the question. The expertise providers have to consent to engage in the communication with the asker to share their expertise. The communication channels used, the contents of the question and answer, the ways the questions is asked and the answers provided, as well as the timings of questioning and asking depend on a set of perceived social variables.

Awareness. Because asking a question implies that the asker is missing some knowledge, the asker needs to take a risk of looking ignorant. Studies show that askers demonstrate different asking behaviors when they are in public or in private or communicating with a stranger or with a friend due to the different levels of feeling psychological safety of admitting the lack of knowledge [3]. Research has also shown that previous social interactions between an asker and a helper leads to easier quality judgment, and helps the interpretation of answers [10].

Access. Social factors in accessing expertise from peers include how and when an asker asks for help from a potential helper. A study has concluded that collocated developers feel socially comfortable to initiate contact because they know each other, know how to approach them, and have a good sense of how important their question is related to what the experts seem to be doing at the moment [8]. Such social cues are heavily used in face-to-face communication through informal interruptions among software development project team members [11]. Rhetorical strategies, linguistic complexity and word choice of the question all influence the likelihood of others responding to a question [10]. Making a personal appeal (e.g. "I need help") in the question results in better and faster responses than making non-personal appeals (e.g. "I have a problem that might be of interest to you") [3]. The expectation of how soon a help would come has been found to be shaped by the history of interactions with the other party [20].

Interruption. Answering, or providing help, consumes the time and attention of the helpers and interrupts their primary task. An interruption is regarded as an unexpected encounter initiated by another person, that disturbs "the flow and continuity of an individual's work and brings that work to a temporary halt to the one who is interrupted" [19].

Collective attention cost. In addition to the cost of the helpers, considerable collective cost could also be incurred. Mailing lists have been heavily used as a means for mediating peer-to-peer knowledge sharing in software development. All the people who have received the question through a mailing list would at least spend some attention about the question before they decide not to answer. When the number of people who receives the question becomes large, the collective attention consumed also becomes considerably large. Attention is quickly becoming the scarcest resource in our society [7].

Social capital. Upon receiving a question, the expert developers need to decide whether and how to engage in collaboration with the asker by expending their precious time and contributing their expertise. This decision is primarily based on their perceived social relationship both with the asker and with the social environment at large. The theory of social capital provides an analytic framework to understand this decision-making process [5]. Social capital is the "sum of the actual and potential resources embedded within, available through, and derived from the network of relationships possessed by an individual or social unit" [14]. It is regarded as important as financial capital and intellectual capital for an individual as well as a social organization because it would promote cooperation and reduce transaction cost [6]. While helping is costly, taking no action also incurs social cost. Saying "no" untactfully to an asker deteriorates the expert's relation with the asker, and affects negatively the expert's social reputation among other peers because it deviates from social norms [18].

4. AN APPOACH: STeP_IN

We have developed the STeP_IN (Socio-Technical Platform for In situ Networking) framework to help developers to use peer expertise based on the above considerations [23]. The goal of STeP_IN is twofold: (1) to increase the ease of accessing peer experts by asking questions, and at the same time (2) to reduce the total cost of experts being interrupted and that of providing help. We try to achieve this goal by creating an ephemeral knowledge network, called a Dynamic Community (DynC) to connect an expertise seeking developer with other developers who have not only technical expertise but also good social relations with the expertise seeker, and support their collaboration with socially aware communication mechanisms.

STeP_IN presupposes a knowledge workspace, which consists of a group of developers, artifacts (their code and related documents), and the three types of relations among them (Figure 3): artifact-artifact, developer-artifact (a developer's *technical profiles*), and developer-developer (a developer's *social profile*). The framework uses those relations to retrieve relevant artifacts for a developer's task at hand, and then to create a DynC for the developer first by identifying experts for the task, and then by selecting experts based on the developer's social profile.

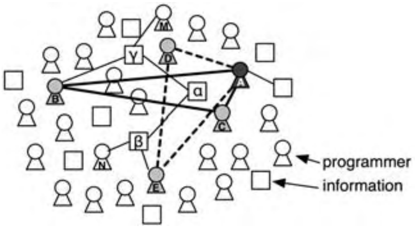


Figure 3: Knowledge Workspace and Relations in STeP_IN

The framework is instantiated in Step_IN_Java (SIJ) for supporting Java developers (see Figure 4) [23]. In SIJ, a Java developer can (1) search for methods, (2) read documents and examples, and (3) ask questions about a specific method to selected experts through the formation of a DynC. See [23] for more details.

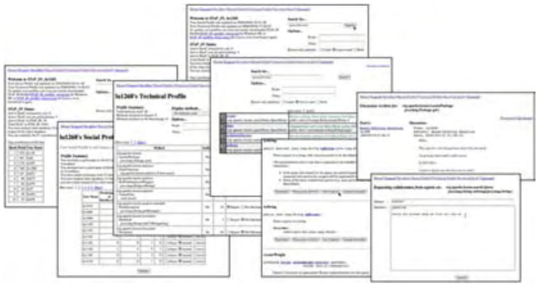


Figure 4: STeP_IN_Java

By using SIJ, developers do not need to have the awareness of who are the experts for the problem that he/she has in seeking for peer expertise. Potential shame of ignorance in asking a question is reduced because only experts with established good relationships are selected. The established social relationships also increase the likelihood for the asker to obtain timely responses because such social relationships are likely to motivate the experts to actively engage in communications with the asker.

A DynC in SIJ complies with the principle of asymmetrical disclosure of information. The membership is not revealed unless one explicitly posts a reply to the DynC. A member, therefore, may leave the DynC (a social equivalent of saying “no”) at any moment without being publicly known. Due to this principle, no participation does not constitute the violation of social norms, which is punishable by the “iron hand of social pressure” of enforcing required individual behavior in a social unit [18]. On the other side, because replying to the DynC reveals the identity of the sender of the message, the DynC members’ contribution is publicly acknowledged and can lead to the improvement of motivation [5].

This socially aware mechanism that allows unwilling peer developers exit socially safely has two implications. The remaining peers are the participants of willing, and hence the expertise sharing becomes more effective. From the perspective of the asker, knowing that other developers could easily exit, he/she feels less pressured to post a question because the availability is controlled by the experts.

Unlike a mailing list, because questions are only sent to DynC members, other developers who have neither interest nor expertise on the topic are not disturbed. The collective cost of attention and interruption is reduced by the reduction of the number of receivers.

5. Summary

This paper analyzed the social factors that affect the knowledge sharing practice during the software development from the perspective of viewing software project as evolving knowledge ecology. The STeP_IN framework was described to support the use of peer expertise with socially aware mechanisms. The framework was illustrated in the SIJ system that supports knowledge collaboration among Java developers.

6. REFERENCES

[1] L.M. Berlin, "Beyond Program Understanding: A Look at Programming Expertise in Industry," in Empirical Studies of Programmers: Fifth Workshop, C.R. Cook, J.C. Scholtz, and J.C. Spohrer, Eds. Palo Alto, CA: Ablex Publishing Corporation, 1993, pp. 6-25.

[2] Cosley, D., Frankowski, D., Terveen, L., Riedl, J., Using Intelligent Task Routing and Contribution Review to Help Communities Build Artifacts of Lasting Value, Proc. CHI06, ACM Press, pp. 1037-1046, 2006.

[3] R. Cross and S.P. Borgatti, "The Ties That Share: Relational Characteristics That Facilitate Information Seeking," in Social Capital and Information Technology, M. Huysman and V. Wulf, Eds. Cambridge, MA: The MIT Press, 2004, pp. 137-161.

[4] Fischer, G., “Symmetry of Ignorance, Social Creativity, and Meta-Design,” Knowledge-Based Systems Journal, Elsevier Science B.V., Oxford, UK, Vol. 13, No. 7-8, pp. 527-537, 2000.

[5] G. Fischer, E. Scharff, and Y. Ye, "Fostering Social Creativity by Increasing Social Capital," in Social Capital, M. Huysman and V. Wulf, Eds., 2004, pp. 355-399.

[6] F. Fukuyama, "Social Capital and Civil Society," presented at IMF Conference on Second Generation Reforms, Washington, DC, 1999.

[7] M.H. Goldhaber, "The Attention Economy," First Monday, vol. 2, 1997.

[8] J. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally-Distributed Software Development," IEEE Transactions on Software Engineering, vol. 29, pp. 1-14, 2003.

[9] J.D. Herbsleb and R.E. Grinter, "Architectures, Coordination, and Distance: Conway's Law and Beyond," IEEE Software, vol. 1999, pp. 63-70, 1999.

[10] I. Illich, Deschooling Society. New York: Harper and Row, 1971.

[11] R.E. Kraut, W. Scherlis, M. Patterson, S. Kiesler, and T. Mukhopadhyay, "Social Impact of the Internet: What Does It Mean?" Communications of the ACM, vol. 41, pp. 21-22, 1998.

[12] T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," presented at Proceedings of International Conference on Software Engineering, Shanghai, 2006.

[13] FD.W. McDonald and M.S. Ackerman, "Expertise Recommender: A Flexible Recommendation System Architecture," Proceedings of CSCW 2000, 2000.

[14] A. Mockus and J. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," in Proceedings of ICSE02. Orlando, FL, 2002, pp. 503-512.

[15] J. Nahapiet and S. Ghoshal, "Social Capital, Intellectual Capital, and the Organizational Advantage," Academy of Management Review, vol. 23, pp. 242-266, 1998.

[16] Nakakoji, K., Ohira, M., Yamamoto, Y., Computational Support for Collective Creativity, Knowledge-Based Systems Journal, Elsevier Science, Vol. 13, No. 7-8, pp. 451-458, December, 2000.

[17] Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., Ye, Y., Evolution Patterns of Open-Source Software Systems and Communities, Proc. IWPSE2002, ACM Press, Orlando, FL, pp. 76-85, May, 2002.

[18] A. Pentland, "Socially Aware Computation and Cmmunication," Computer, vol. 38, pp. 33-40, 2005.

[19] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," IEEE Transactions on Software Engineering, vol. 30, pp. 889-903, 2004.

[20] A.M. Szoestek and P. Markopoulos, "Factors Defining Face-to-Face Interruptions in the Office Environment," in Proceedings of Conference on Human Factors in Computer Systems, 2006, pp. 1379-1384.

[21] J.R. Tyler and J.C. Tang, "When Can I Expect an Email Response? A Study of Rhythms in Email Usage," in Proceedings of the Eighth European Conference on Computer Supported Cooperative Work (Ecscw2003). Helsinki, 2003, pp. 239-258.

[22] Wenger, E., Communities of Practice − Learning, Meaning, and Identity. Cambridge, UK: Cambridge University Press, 1998.

[23] C. Westrup, "On Retrieving Skilled Practices: The Contribution of Ethnography to Software Development," in Social Thinking: Software Practice, Y. Dittrich, C. Floyd, and R. Klischewski, Eds. Cambridge, MA: MIT Press, 2002, pp. 95-110.

[24] Y. Ye, Y. Yamamoto, K. Nakakoji, Helping Programmers through In Situ Networking of Peer Expertise, ICSE 2007 (submitted).

Distributed cognition in software engineering research: Can it be made to work?

Jorge Aranda
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4, Canada
1-416-946-8878
jaranda@cs.toronto.edu

Steve Easterbrook
University of Toronto
40 St. George Street
Toronto, Ontario, M5S 2E4, Canada
1-416-978-3610
sme@cs.toronto.edu

ABSTRACT

Distributed cognition is a theoretical and methodological framework that considers social groups, their artifacts, and their contexts as a single cognitive entity working towards the solution of a shared problem. In this paper we briefly describe the framework and consider its strengths and weaknesses as a theoretical foundation for software engineering research. We propose a series of techniques to address the methodological problems that the application of the framework entails in our research field. Finally, we present an ongoing exploratory case study that aims to evaluate the adaptability of the framework and of the techniques we propose here.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *programming teams, software process models.*

General Terms

Documentation, Experimentation, Human Factors, Theory.

Keywords

Distributed Cognition, Social Networks Analysis, Artifact Analysis, Empirical Software Engineering.

1. INTRODUCTION

Research of software engineers at work –of their team structures, interactions, and dynamics– has been largely performed as a butterfly collection exercise: We have many interesting bits of results, but we do not have a theoretical framework that links the separate phenomena we observe, unifies our perspectives of the domain, and allows us to generate testable predictions of software projects and software teams. As a consequence, our findings are not exploited to their full potential; and our research effort is often spent exploring unviable or shallow hypotheses [6].

For illustration purposes, consider the extensive literature on design and code inspections [7]. Although there have been dozens of studies testing the phenomenon, they provide little insight as to why it occurs, how can its beneficial effect be amplified, and what could possibly be the consequence of performing inspections in ways that have not been empirically tested. The reason, we claim, is that until recently inspection studies were not designed over a theoretical foundation that predicted their effects and addressed these issues. If inspection researchers had a theory to guide their work, they could have spent their efforts validating it and probing its predictive power, yielding even stronger findings for our domain.

The inspections literature is the norm, not the exception, when it comes to theory building and theory validation in the software engineering realm. To address this problem, we are evaluating the capabilities of a theoretical framework (distributed cognition) and its applicability to software engineering in an exploratory case study. In this paper we present the gist of the distributed cognition theory, its strengths and weaknesses with regards to software engineering research demands, and the adaptations we feel are necessary for such a framework to be convenient for our research. We also briefly describe the case study we are conducting and the roadmap we intend to follow in the near future.

2. DISTRIBUTED COGNITION

Distributed cognition is an interdisciplinary theoretical framework designed to study cognition as it occurs in socially situated contexts. Its unit of analysis is the functional system of people and artifacts in charge of executing a cognitive task. That is, for a distributed cognition researcher, the functional system is a single cognitive entity, and although no element within this entity may know how to solve the cognitive task, the full range of interactions and transformations of information within the group produce a workable solution to the cognitive problem at hand [11]. Perhaps the classic example of distributed cognition research is Hutchins' study of sea navigation [4], where each person in the navigation team of a ship performs a set of simple tasks based on their role and the information available to them, and although no person in the team has a full knowledge of the situation, the end result is a calculation of the ship's position in the world.

Although the distributed cognition framework is too extensive to be summarized here, there are several properties about it worth mentioning. First, it centers on the study of situated cognitive activities, as opposed to artificial laboratory settings. According to the theory, cognitive performance should not be analyzed in constrained settings, since much of people's real cognitive work is done by the interaction among them and with their context.

Second, artifacts are viewed as embodied knowledge –they store rules and processes that simplify the cognitive tasks of their users. Therefore, analyzing the artifacts people use is an essential aspect of the framework.

Third, identifying the paths that chunks of information follow to reach the persons that need them is a key consideration of distributed cognition work. Team members that work on a cognitive problem start up with different bits of knowledge, and an important step towards solving the problem is to share and transform them, through mediated or direct communication, until they reach the person who needs them.

Finally, the framework studies cognitive work on two different levels: In the short term, it focuses on the actual resolutions of cognitive problems; while in the long term, it analyzes the learning and structuring activities that take place in teams.

Since its original formulation, the framework has been used to examine a wide variety of groups and contexts, including navigation [4], aviation [5], hotline centres, rescue teams, and, in one occasion, software developers performing maintenance tasks [1]. Unfortunately, so far there have only been a few teams applying the framework and producing this research –most notably Hutchins' own research group at San Diego.

The distributed cognition framework is still far from being generally accepted by any research community. In the CSCW literature, a response by Bonnie Nardi to a paper on theories for CSCW [3] critiques several theoretical and practical problems of the framework [10]. She points out how its insistence on ethnographic methods, and in particular of ethnomethodology, causes an "anemic theoretical development", which, she warns, leads to "a withering of community in any field of study." She also notes that distributed cognition, as proposed by Hutchins and in parallel to ethnomethodology, is suspicious of conceptual elaboration, undermining communication and comprehension efforts in the research community.

3. DISTRIBUTED COGNITION IN SOFTWARE ENGINEERING

3.1 Applicability of the theory: Benefits and drawbacks

The idea of conceptualizing software development as a socially distributed, artifact-intensive cognitive activity is compelling, and we believe the software engineering field could reap important benefits by adopting this view. Here are some of the advantages that result from appropriating this theoretical foundation:

- A systemic view of software teams, which includes the social aspects of team collaboration and the study of the interactions between humans and their artifacts. All software development practices, documents, and tools, can be re-interpreted and explored within this view.
- An abstraction of all interactions and uses of artifacts as *transformations of representational states across representational media* [4], which allows for evaluating the effectiveness of alternative transformations by interpreting software development techniques (such as code reviews, pair programming, and prototyping) as transformations and representations of information with particular coordination- and communication-related strengths and weaknesses.
- An emphasis on analyzing artifacts both as embodied knowledge and as communication media, leading to insights about new and modified proposals for tools and languages to capture and transfer that knowledge.
- A consideration of individual and organizational learning, role specialization dynamics, and the context in which these phenomena take place, which may prove to be a fruitful perspective for software project management research.

Both in general, as a paradigm of the software development field, and in particular, as a collection of techniques for improving the context and tools in which cognitive-intensive activities take place, distributed cognition seems to be a useful perspective to adopt for software engineering research. However, if it is to become a theoretical foundation for this research, it will need to undergo significant methodological alterations to achieve practicality.

We think software engineering research cannot be built over an ethnomethodological foundation. Ethnomethodological studies are necessarily constrained to the analysis of particular, detailed phenomena, and the amount and variability of such phenomena in software projects is overwhelming, even for small-scale projects. It boggles the mind to consider how a comprehensive ethnomethodological study, of the kind performed in the distributed cognition literature, could be carried out in a large-scale, geographically distributed, multi-year development project.

To turn the framework into a feasible alternative for this type of research, we need methods that abstract away some of the details of day-to-day phenomena and focus on detecting the essential patterns of communication, team structure, and artifact use in software projects. Before proposing any methods, however, we must address the question of whether such departures from ethnomethodology are compatible with the core ideas of distributed cognition or, alternatively, ethnomethodological detail is an essential component of the framework.

It seems to us that ethnomethodology is, though valuable, accidental to the theory; a result of the background of the original distributed cognition researchers. Just as it might be desirable for cognitive scientists (but impractical under our technological and practical circumstances) to examine every synapse in the brain, analyzing every utterance of a problem-solving group is not essential to the conceptualization of such group as a distributed cognitive entity.

What, then, is essential? To get a basic picture of a distributed cognitive system, at least the following elements need to be analyzed:

- Group structure and patterns of group interaction
- Artifacts (tools, documents), and patterns of artifact use
- Nature and frequency of tasks

- Development of shared understanding, breakdowns and recoveries

There are techniques, both from distributed cognition and from other disciplines, to study these types of information. In the next subsection we propose some of the most promising ones.

3.2 Social Network Analysis (SNA)

Sociologists have developed a collection of methods to analyze the structural and dynamic qualities of social groups [13]. We do not have the space to describe them in detail, but we would like to mention a short list of them. To start, social network graphs and simple SNA measurements such as *centrality* and *density* provide an initial overview of the structure of a group. More elaborate techniques, such as *blockmodelling* (for clustering nodes based on their similarities in several networks) and *positional analysis* (for simplifying the information in network data sets), among others, complete the picture of group structures. Finally, other SNA-inspired concepts, such as knowledge transfer and social capital, add fruitful perspectives to the study of group interactions.

Some kinds of software projects are particularly amenable to SNA methods –those for which communication takes place almost exclusively in electronic form, such as most open source projects [9]. In these cases, a full record of interactions is available to the researcher, and one can track the proposal of new ideas, the types and frequency of contributions, and the transfer of information among project members. For other projects, particularly those in which participants are collocated, many exchanges of information and much knowledge of the social structure of the group is not recorded electronically or in the project's documentation, and must be extracted directly from participants.

However, an important advantage of SNA methods for our purposes is that the data they require are relatively easy to collect. Conducting case studies to understand the full structure of software development teams becomes feasible, and surveys of wide ranges of software houses are also possible.

On the other hand, SNA methods were designed for sociological goals, and they are often concerned with topics that are not of immediate relevance to software engineering, such as power relations, social support, and the job market. To our knowledge, no study has yet analyzed in detail the implications of applying the methods of SNA to the software engineering field.

3.3 Artifact analysis

Some of the most satisfying results from distributed and external cognition studies are their analyses of artifacts people use to perform their tasks. Through these analyses we discover how cognitive activities are simplified by representing information and rules "in the world", rather than in people's heads [15], and by re-representing complex information in ways that simplify its understanding [12].

For software development projects, artifact analyses may provide insight into the efficacy and dynamics of document and tool use. For documents (a category in which we include, for instance, specifications, models, and emails), the researcher may find what information flows among people, how expressive, efficient, and useful are the representations, how quickly do they become obsolete or out of sync with the world, and what are the skills necessary to create them, modify them, and read them.

The thorough study of all documents used in a project is not practical. But collecting data on the frequency with which different types of documents are used and their relevance for each group member provides us with useful patterns of interactions and of team dynamics. It will also point to particularly relevant documents, which may be studied with the more careful detail that traditional distributed cognition literature displays.

For tools, of which every programming language, IDE, project website, and debugger are examples, the researcher may uncover cognitive benefits provided by new and existing proposals based on the computational effort they demand from their users.

Tool analyses are detailed and time-consuming. However, once performed, their findings are applicable for projects that use the same tools under similar settings, paying off the investment considerably.

3.4 Other approaches

We are evaluating the utility and practicality of other approaches to support distributed cognition in software engineering; approaches that in principle can be effective complements to SNA and artifact analysis, but whose empirical validity is still not clear.

One such alternative is *conceptual sketching* [2], which may provide rich details about the networks, perceptions, and mental models of participants of a software team. Conceptual sketching, however, may also be prone to misinterpretations and vague results, which are, of course, undesirable characteristics in software engineering research.

4. CASE STUDY

To test the viability of the distributed cognition framework and the methodological adaptations we propose, we are conducting a pilot case study on the release team of a software division at IBM. This work feeds upon other studies of developers, such as that of LaToza et al. [8], and other attempts to conciliate software engineering and distributed cognition [14].

The release team is a high-impact, high-interaction volume group within the division. It oversees product development and serves as a bridge between "technical" and "business" people. This bridge role requires from them, in addition to advanced project management skills, a familiarity with at least two different professional cultures, vocabularies, and goals. They are focal enablers of shared understanding in the division, in the sense that they are the main point of contact for developers to learn project requirements, and for managers to learn their projects' status.

For these reasons, the people at the release team have experienced the need to create roles, team dynamics, and processes that help them handle their responsibilities and coordinate the efforts of the full division towards shipping their releases. We think the analysis of these roles, dynamics, and processes, with a distributed cognition lens, should be particularly insightful.

We designed our case study to explore these phenomena. We decided to interview every member of the team with a structured questionnaire that probes the techniques we described above. Our interview has four main sections. First, we ask participants to draw conceptual sketches of their team, of their interactions with other teams, and of their division within and outside the company. Second, we collect social network data, focusing on several types of personal networks (information consumers and producers, collaborators, mentors, and informal networks). Third, we ask participants to describe the main activities they perform according to their role, and to list the artifacts (documents and tools) that they use to perform each of these activities. Finally, we ask them open-ended questions about the goals of their role and their team, success criteria, success factors, and an overall description of their position in the company.

Each section of the interview will first be analyzed separately, and their findings will later on be put together to detect patterns among them. We designed the questionnaire in a way that allows us to evaluate both the team itself and the methods we chose to use, so we can refine them for future larger-scale case studies.

We are, at the moment of writing, in the data collection phase of our case study. We have collected the data of nine participants, with five more to go. We will proceed to analyze their conceptual sketches and their social networks data separately, and to identify the most relevant tools and documents they use in order to perform an artifact analysis on them.

After refining our techniques with findings of this case study, we plan to conduct at least two other studies in the same organization. The first is an extension of our current study – including data from the technical and business groups that interact with the release team we are analyzing. The second is a replication of our initial study, for a different release team, in an effort to detect the patterns that arise from two divisions with different cultures within the same corporation.

As an end result of these empirical studies we expect to obtain two types of benefits: For the organization, we should be able to produce recommendations for tool, document, and process improvements. For our research team, we will have data regarding the viability of the methodological approaches we describe in this paper, and the adaptations we find necessary for their successful implementation by our research community.

5. CONCLUSIONS

Distributed cognition is a fruitful foundation to support research of software engineers at work, but if it is to be used for this purpose, we need to overcome its methodological constraints with alternatives such as the ones discussed in this paper. We believe that, by rejecting the notion that we can (or should) capture and analyze every detail of the interactions of developers, software engineering research can benefit greatly from the perspectives the theory provides while allowing studies of the social side of software development to remain feasible.

We think that the methods and techniques we described above can support empirical studies of this kind by substituting the ethnomethodological studies of traditional distributed cognition with workable solutions that still enable us to make key findings. However, we do not have any data to back up these claims yet. Our pilot case study, and possible subsequent studies, will allow us to make an evaluation of which of the techniques we propose are off the mark, which need some adaptation, and which work well for our field.

6. ACKNOWLEDGMENTS

We would like to thank and acknowledge the financial support from Bell University Labs at U of T. We are also very grateful to IBM for supporting and cooperating with our study.

7. REFERENCES

[1] Flor, N.V., and Hutchins, E. Analyzing Distributed Cognition in Software Teams: A Case Study of Collaborative Programming During Adaptive Software Maintenance. *In Koenemann-Belliveau, J., Moher, T., and Robertson, S. (Eds), Empirical Studies of Programmers: 4th Workshop,* 1992.

[2] Goel, V. *Sketches of thought.* Cambridge, MA: MIT Press. 1995.

[3] Halverson, C.A. Activity Theory and Distributed Cognition: Or what does CSCW need to DO with theories? *Computer Supported Cooperative Work, 11,* 243-267, 2002.

[4] Hutchins, E. *Cognition in the Wild*. MIT Press, Cambridge, MA, 1995.

[5] Hutchins, E. How a Cockpit Remembers Its Speeds. *Cognitive Science, 19,* 265-288, 1995.

[6] Jørgensen, M., and Sjøberg, D. Generalization and Theory-Building in Software Engineering Research. *Proceedings of the Workshop on Empirical Assessment in Software Engineering (EASE'04),* 2004.

[7] Laitenberger, O., and DeBaud, J.M. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software, 50, 1,* 2000.

[8] LaToza, T., Venolia, G., and DeLine, R. Maintaining Mental Models: A Study of Developer Work Habits. *Proceedings of the International Conference on Software Engineering (ICSE'06).* (Shanghai, China, May 20-28, 2006).

[9] Madey, G., Freeh, V., and Tynan, R. The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory. *8th Americas Conference on Information Systems,* 2002.

[10] Nardi, B.A. Coda and Response to Christine Halverson. *Computer Supported Cooperative Work, 11,* 269-275, 2002.

[11] Rogers, Y., and Ellis, J. Distributed Cognition: an alternative framework for analysing and explaining collaborative working. *J. of Information Technology, 9, 2,* 119-128, 1994.

[12] Scaife, M., and Rogers, Y. External cognition: how do graphical representations work? *International Journal of Human-Computer Studies, 45,* 185-213, 1996.

[13] Wasserman, S., and Faust, K. *Social Network Analysis: Methods and Applications.* Cambridge U. Press, 1994.

[14] Ye, Y. Supporting Software Development as Knowledge-Intensive and Collaborative Activity. *Proceedings of the 2nd Intl. Workshop on Interdisciplinary Software Engineering Research (WISER'06).* (Shanghai, China, May 20-28, 2006).

[15] Zhang, J., and Norman, D. Representations in Distributed Cognitive Tasks. *Cognitive Science, 18,* 87-122. 1994.

D-SNS: A Knowledge Exchange Mechanism Using Social Network Density among Mega-Community Users

Masao Ohira[1]
[1]Department of Information Science,
Nara Institute of Science and Technology
8916-5 Ikoma, Nara, 630-0192, Japan
masao@is.naist.jp

Kumiyo Nakakoji[2, 3]
[2]RCAST, University of Tokyo
4-6-1 Komaba, Meguro,
Tokyo, 153-8904, Japan
[3]SRA Key Technology Lab. Inc.
kumiyo@kid.rcast.u-tokyo.ac.jp

Ken-ichi Matsumoto[1]
[1]Department of Information Science,
Nara Institute of Science and Technology
8916-5 Ikoma, Nara, 630-0192, Japan
matumoto@is.naist.jp

ABSTRACT

SourceForge.net (SF.net) is a large-scale online community hosting a number of open-source projects. We regard SF.net as a "*mega community*" because many of the members of SF.net are members of one or more open-source projects, which themselves form online communities. By regarding SF.net as a mega online community, we have identified four types of social relationships among the members of SF.net. This paper describes a mechanism to exploit measurement of the density of social networks to understand the nature of each type of social relationships in the mega community. We present D-SNS (Dynamic Social Networking System), which promotes knowledge exchange among SF.net users, by using measurement results of the density of social networks.

Categories and Subject Descriptors

H.5 [**Information Interfaces and Presentation**]: Group and Organization Interfaces – *collaborative computing, computer-supported cooperative work, organization design, web-based interaction*

General Terms

Management, Measurement, Human Factors

Keywords

Mega community, knowledge exchange system, social network analysis, density of social networks

1. INTRODUCTION

An online community such as open source software (OSS) community is a means to create artifacts by collaboration among community members. A number of studies have tried to understand various aspects of online communities [1][2][3]. The primary objectives of most of the studies are to reveal characteristics of collaboration and communication in a single community, and to construct methodologies and tools for supporting online communities. An increasing number of recent studies especially focus on social relationships among community members using social network analysis (SNA) in order to support activities of members in the community [4][5].

This paper presents our study that focuses on social relationships among people in a particular kind of online community, a "*mega community*." A mega community is a large-scale online community consisting of a number of smaller-scale communities. The interesting aspect of a mega community is where a member seems to naturally switch the role between a member of a mega community and a member of a smaller community. In fact, many of the members of a mega community we studied belong to more than one smaller scale communities [6]. As the result, there are different kinds of multiple social relationships in a mega community. The goal of our study is to support creation of social relationships suitable to various roles of members in a mega community.

In the next section, we illustrate SourceForge.net[1], which is an online OSS development community, as a representative example of a mega community. We then present the particularity of social relationships observable in SF.net. Section 3 describes a way to understand the nature of social relationships by measuring the density of social networks. Section 4 discusses how we can use measurement results of the density of social networks. We introduce the prototype system called D-SNS (Dynamic Social Networking System), that promotes knowledge exchange among SF.net users, as an application of exploiting measurement results of the density of social networks.

2. SourceForge.net: A MEGA COMMUNITY

SourceForge.net (SF.net) is a large-scale online community for OSS (Open Source Software) development. People register in SF.net to become a SF.net user. The SF.net users have a variety of roles (e.g., developers, bug reporters, end-users, donators and so on). The number of unique user accounts is over 1.2 millions in March 2006.

We regard SF.net as a "mega community," which is a community for a number of OSS communities. An OSS community in SF.net is called a "project". Over one hundred thousand OSS projects are registered on SF.net. SF.net users participate in activities of each OSS project such as releasing OSS, discussing issues on OSS development, reporting bugs and so forth.

The social relationships among SF.net users are created through the activities. SF.net users communicate with each other by using mechanisms such as bulletin board systems called forums, mailing lists, and bug reporting (tracking) systems. They rarely meet face-to-face. Here, social relationships are assumed to be

[1] SourceForge.net, http://sourceforge.net

relations emerged from results of communications among SF.net users.

There are discriminative social relationships in a mega community such as SF.net, which are different from social relationships in a single, common OSS community. Figure 1 simply illustrates four kinds of social relationships in SF.net as graphs called social networks. Graphs for social networks represent persons as nodes and relations between persons as lines (edges).

Figure 1-(a) depicts the social relationships among all SF.net users in case of viewing SF.net as a single community. The social relationships are defined as the same as that in a common OSS community.

Figure 1-(b) shows the social relationships in each project in SF.net. A small circle represents an OSS project as a single community. The social relationships are defined by relations created in each project.

Figure 1-(c) represents the social relationships which user(X) has in SF.net, in user(X)'s point of view. This type of social network is called ego-centric network. User(X) has connections to five users.

Figure 1-(d) shows the social relationships which user(X) has in each project in SF.net, in the user(X)'s point of view. Because user(X) participates in three projects, her/his social relationships are represented as three social (ego-centric) networks.

In this way, different types of social relationships exist in a mega community depending on ways of cutting off social relationships, that is, standpoints of people involved in the mega community (e.g., (a) is for administrators of a mega community, (b) is for managers of communities, and so on.). It would be important for people in a mega community to understand the nature of social relationships according to own roles or positions.

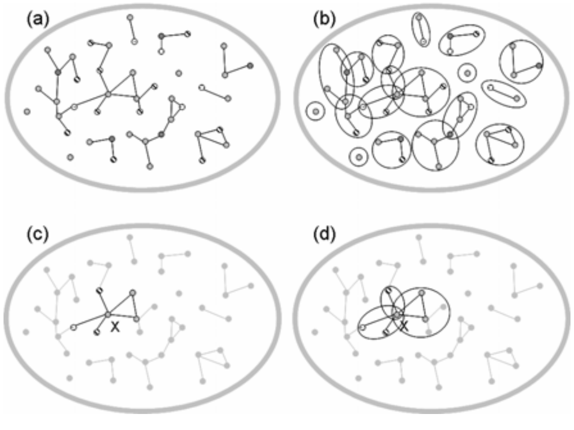


Figure 1. Four types of social relationships in a mega community

3. MEASURING SOCIAL RELATIONSHIPS

3.1 Characteristics of Social Relationships

There are two social networks extremely representing characteristics of social relationships. One is that called open network in the left of Figure 2. The other is closed network in the right of Figure 2.

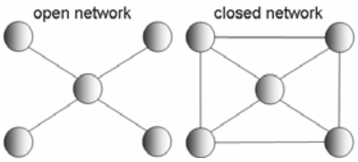


Figure 2. Open network and closed network

The characteristics of open network and closed network are described by [7] as follows.

> ***Open Network****: is a large, open, diverse, and externally focused network. It is excellent for getting lots of new information, learning about new opportunities, and finding resources. However, it is not so good for building consensus, producing consistent expectations, or developing a sense of common mission (may be prone to conflicts or tensions).*
>
> ***Closed network****: is a small, closed, homogeneous, and internally focused network. It is good for building group loyalty, identity, and a sense of common purpose. The disadvantage is that it may be inadequate for getting information or other resources, or insufficient for influencing people outside the networks. It is subject to group thinking and the development of an us-versus-them view of the world.*

3.2 Measuring Network Density

There are many metrics in the area of social network analysis [8][9], which can be used to know characteristics of social relationships in an organization or group. Measuring the density of social networks is a simple way to know whether social networks have the characteristics of open network or closed network [9]. If social networks with low density, the social networks tend to have the characteristics of open network. If social networks with high density, the social networks often have the characteristics of closed network.

The density of social networks is defined as the number of lines (edges) in social networks, expressed as a proportion of the maximum possible number of lines [8]. The formula for the density of social networks is

$$ND = \frac{2l}{n(n-1)}$$

where l is the number of lines (edges) in the networks and n is the number of nodes in the networks. The values of ND (network density) can be from 0 to 1.

As we described in Section 2, four types of social relationships exist in SF.net. Here, the network density can be defined for each type of social relationships.

(a) $ND(SF)$: ND in SourceForge.net
(b) $ND(P_i)$: ND in each project(P_i) in SF.net
(c) $ND(U_j, SF)$: ND of an ego-centric network each user(U_j) has in SF.net
(d) $ND(U_j, P_i)$: ND of an ego-centric network each user(U_j) has in each project(P_i) in SF.net

Table 1. Four types of network density in SF.net (Nov. 2005)

	ND	Max./Min.	σ^2
(a) *ND(SF)*	0.15 x 10^{-10}	N/A	N/A
(b) *ND(P_i)*	0.24 (avg.)	1.0/0	0.13
(c) *ND(U_j, SF)*	0.65 (avg.)	1.0/0	0.16
(d) *ND(U_j, P_i)*	0.66 (avg.)	1.0/0	0.17

No one ideal ND can fit all people in a mega community. Each ND is an indicator to understand the current state of social relationships in a community or around own, and to think how the social relationships ought to be in the future.

4. USING NETWORK DENSITY

The four types of *ND*s can be used to design support tools suitable to various roles or positions of people in a mega community.

For instance, for administrators of SF.net and managers of projects, *ND(SF)* and *D(P_i)* are respectively important clues to know the state of social relationships among users in SF.net and projects they have to manage. If a project manager thinks his project should be more closely united than the current, he would need tools for mediating or facilitating communications among his project's members, so that *ND(P_i)* will be higher than the current.

For each users in SF.net, measuring *ND(U_j, SF)* and *ND(U_j, P_i)* would be helpful to support them. If user(U_j) have social relationships expressing as high *ND(U_j, SF)*, she might want a help for finding people with whom she have never communicate before because she cannot get new information through her current social networks. A user who often has a conflict with other users in a project might hope *ND(U_j, P_i)* will be more higher.

Table 1 shows the result of analysis on four *ND*s in SF.net. The data for calculating four *NDs* is communication logs accumulated in forums (bulletin board systems). Using forums, SF.net users (e.g., developers, end-users, bug reporters, and so on) discuss issue related to OSS development. If user(U_A) posts a message in a forum for project(P_i) and user(U_B) replies the message, then it assumes that there is a social relation between user(U_A) and user(U_B) in project(P_i). We collected all messages in all accessible forums (1,230,000 communication logs among 160,000 SF.net users in 90,000 projects) and extracted social relationships among the users.

From the result in Table 1, *ND(SF)* is extremely low compared to other *NDs*. In case of viewing SF.net as a community, the social relationships among SF.net users are not close at all. In contrast to *ND(SF)*, the average of *ND(P_i)* is much higher (0.24). The result is natural because OSS development in SF.net proceeds through project-based activities and cross-project OSS development is infrequent [6]. Both *ND(U_j, SF)* and *ND(U_j, P_i)* show furthermore high values. This is because over 80% of all projects in SF.net consist of less than 3 developers [6].

The density of social network, which represents the nature of the social relationships in a mega community, varies according to where we are looking at in a mega community. Therefore, in order to support to build social relationships in a mega community, we would need to design tools in consideration of which aspects of social relationships we are trying to support.

5. D-SNS: AN APPLICATION

This section introduces the prototype system called D-SNS (Dynamic Social Networking System) that promotes knowledge exchange using social relationships among SF.net users, as an application of exploiting measurement results of the density of social networks. In this case, *ND(Uj, SF)* is considered to design the system. The detail of D-SNS is described in [6].

D-SNS collects communication logs in all accessible forums in SF.net and extracts information on social relationships among SF.net users, information on technical terms used for finding knowledgeable developers (i.e., information on who is knowledgeable about what), and information on communication frequency among users, from the communication logs.

D-SNS helps a user chose whom she should communicate with, according to the state of the user's social relationships. If a system's user inputs a question related to OSS into the system, the system finds other users knowledgeable on the question and recommends knowledgeable users who answer the question.

5.1 Mechanism

Figure 3 shows the mechanism of D-SNS. Here, suppose that user(*U_A*) with *ND(U_A, SF)*=0.7, who wants to make *ND(U_A, SF)* more lower, is asking a question. At first, user(*U_A*) inputs a question into D-SNS.

5.1.1 Searching knowledgeable people

D-SNS searches knowledgeable users using results of keyword matching between technical words stored in the system and keywords user(*U_A*) input. The system selects up to 20 users and delivers the questions to them.

5.1.2 Sorting by network density

If some users answer the question, D-SNS calculates *ND(U_A, SF)* after communicating with the users as Figure 4, and sorts the

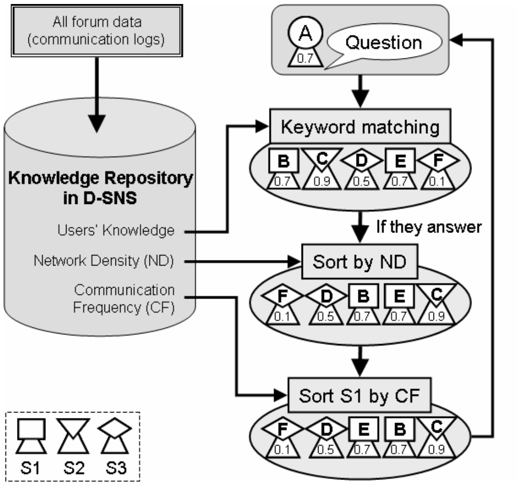


Figure 3. Mechanism of D-SNS

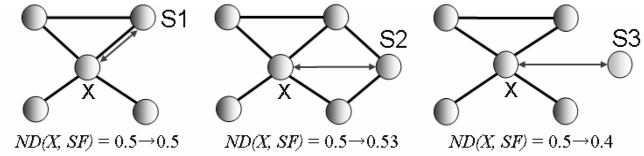


Figure 4. A communication partner and changes of network density

users in order of lower $ND(U_A, SF)$ in this case. Ordering depends on each user(U_j)*'s* preference.

5.1.3 *Sorting by communication frequency*

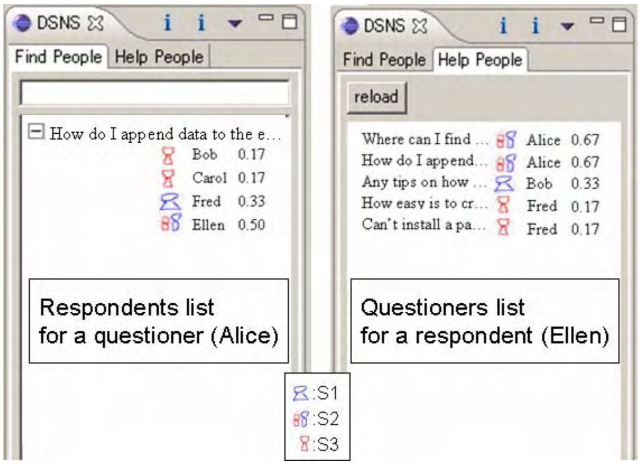
If $ND(U_A, SF)$ after communicating with knowledgeable users is same such as user(U_B) and user(U_E) in Figure 4, D-SNS sorts the users in order of higher communication frequency because user(U_E) who often communicates with user(U_A) might have better understandings of user(U_A)'s questions or demands than user(U_B).

5.2 User Interface

Figure 5 shows the user interface of D-SNS. The left of Figure 5 is for questioners (Alice). Alice can ask a question from the "Find People" tab. If someone replies the question, a list of respondents will appear as the list in the left of Figure 5.

The icons mean that S1 is a user within 1 degree of separation from Alice (i.e. have communicated with S1 before) as in the left of Figure 4, S2 is a user within 2 degrees from Alice as in the middle of Figure 4, and S3 is a user with more than 3 degrees of The numbers of the right of icons shows $ND(U_{Alice}, SF)$ if Alice communicates with the listed users.

In a similar way, a respondent (Ellen) can find questioners who would like to know Ellen's knowledge from the "Help People" tab in the right of Figure 5. If Ellen wants to tell a questioner (suppose Alice) something Ellen knows, Ellen can reply to Alice' questions using BBS only accessible for Ellen and Alice. If Ellen does not reply any questions, none of questioners can know that because questioners cannot know their questions will be delivered to whom.



How do I append data to the e...
Bob 0.17
Carol 0.17
Fred 0.33
Ellen 0.50

Respondents list for a questioner (Alice)

Where can I find ...	Alice	0.67
How do I append...	Alice	0.67
Any tips on how ...	Bob	0.33
How easy is to cr...	Fred	0.17
Can't install a pa...	Fred	0.17

Questioners list for a respondent (Ellen)

:S1
:S2
:S3

Figure 5. User interface of D-SNS

6. FUTURE WORK

In the near future, we need to enlarge data sources (e.g., mailing lists and bug reporting systems) for extracting social relationships and users' knowledge. We have a plan to elaborate ways of calculating *NDs* by using directed graphs or weighted graphs. We also would like to design support tools for administrators and managers in a mega community using $ND(U_j, SF)$ and $ND(U_j, P_i)$.

7. ACKNOWLEDGMENTS

This work is supported by the EASE (Empirical Approach to Software Engineering) community in the Comprehensive Development of e-Society Foundation Software program and Grant-in-aid for Scientific Research (B) 17300007, 2006 and for Young Scientists (B), 17700111, 2006, by the Ministry of Education, Culture, Sports, Science and Technology of Japan.

8. REFERENCES

[1] Beenen, G., Ling, K., Wang, X., Chang, K., Frankowski, D., Resnick, P. and Kraut. R. E. Using social psychology to motivate contributions to online communities. In *Proceedings of the conference on Computer supported cooperative work (CSCW'04)*, 2004, 212-221.

[2] Millen, D. R. and Patterson, J. F. Stimulating social engagement in a community network. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'02)*, 2002, 306-313.

[3] Mockus, A., Fielding, R. and Herbsleb, J. D. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3), 2002, 309-346.

[4] Saltz, J. S., Hiltz, S. R., and Turoff, M. Student social graphs: visualizing a student's online social network. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'04)*, 2004, 596-599.

[5] Stefanone, M., Hancock, J., Gay, G., and Ingraffea, A. Emergent networks, locus of control, and the pursuit of social capital. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'04)*, 2004, 592-595.

[6] Ohira, M., Ohoka, T., Kakimoto, T., Ohsugi, N. and Matsumoto, K. Supporting Knowledge Collaboration Using Social Networks in a Large-Scale Online Community of Software Development Projects. In *Proceedings of APSEC2005 Workshop on Supporting Knowledge Collaboration in Software Development*, 2005, 835-840.

[7] Baker, W. E., Achieving Success through Social Capital. John Wiley & Sons Inc., 2000.

[8] Wasserman, S. and Faust, K. Social Network Analysis: Methods and Applications. Cambridge University Press, 1994

[9] Scott, J., Social Network Analysis: A Handbook. SAGE Publications, 2000.

Collaborating Over Project Schedules

Suzanne Soroczak[1,2] and David W. McDonald[2]
[1]Information Systems and Technology Group, Intel Corporation, Hillsboro, OR 97124
[2]The Information School, University of Washington, Seattle, WA 98195
suzanne.m.soroczak@intel.com, dwmc@u.washington.edu

ABSTRACT

Numerous case studies and ethnographies have shown project management in software engineering to be a collaborative activity. However, project management "tools of the trade" do not readily support collaboration. As a result, project management breakdowns can occur. This paper discusses the issues of collaborative project management and makes recommendations for future project management tool development.

General Terms

Management, Documentation, Standardization

Keywords

Software Engineering, Project Management, Collaboration, Project Teams, PERT Chart, Gantt Chart, Work Breakdown Structure (WBS).

1. INTRODUCTION

Software development entails many activities including requirements gathering, design, coding, documenting, testing, and debugging. One activity that weaves together the entire process is project management. Many different project management methodologies have been developed, tried, and documented. Yet the typical tools of project management have remained largely unchanged. In this paper we examine project management practices in software development teams as reflected in the prior literature. This paper reports on a survey and analysis of case studies and ethnographies of software development teams from a project management perspective.

In the next section we provide a brief introduction to current project management tools and practices. Next we discuss a number of project management breakdowns identified in the literature on software development teams followed by the social aspects of managing and planning software development in a globally distributed environment. In section four, we outline issues with implications for the development of a new generation of project management tools. We close with ideas for future research in distributed collaborative project management.

2. PROJECT MANAGEMENT PRACTICE

The Project Management Institute (PMI) defines project management as "the art of directing and coordinating human and material resources throughout the life of the project by using modern management techniques to achieve predetermined objectives of scope, cost, time, quality, and participant satisfaction."[17] The PMI and the many practitioners of project management have built a book of knowledge (PMBOK) or "best practices", which aims to guide project managers in the art of managing projects. The best practices are meant to be a general guide to practice, applicable in many domains. In software development, project management methodologies are seen as one of the software development models used in conjunction with one or more management methods and techniques [18]. Developing a project task list and schedule is viewed as project management *practice* [16].

2.1 Project Management Tools Of The Trade

Project management tools are used for planning, scheduling, tracking, and controlling projects – the essential activities of managing a project. The most common project management tools are the project task list and the project schedule. A project task list is an enumeration of individual tasks and subtasks to be completed for the project. A project schedule is "a permanent record of a set of tasks to be executed, along with their predicted durations and completion times."[20] A schedule usually contains task attribute data and specifies who has responsibility for each task and information about the dependencies between tasks.

In Whittaker and Schwarz's paper on scheduling mediums, they identify the important functions that schedules serve. Schedules serve as a joint to-do list, allowing people to coordinate future action. They can be seen as a type of contract of the work promised to be executed. Schedules also "provide information without the overhead of interrupting other team members or calling a group meeting, serve as an external communication tool to people outside of the group, and can assist individuals in organizing their own work."[20]

Project teams make use of a variety of software tools for creating and maintaining project tasks lists and schedules. Fox and Spence's survey of project managers identified the top "tools of the trade" in use in software companies as MS Project, Project Workbench, and MS Excel [8]. Interestingly, project managers rely on nontraditional project management tools, such as MS Excel, as readily as more project-focused tools. This is especially true for projects of a short duration (less than six months), where the start-up costs associated with project-focused tools may dissuade people from their use.[8] As well, emergent technologies like SharePoint, Blogs, and Wikis are being adopted for *ad hoc* project organization and communication [11].

2.2 Creation of the Project Schedule

According to the PMBOK, a project schedule begins with the work breakdown structure (WBS). The WBS is an enumeration of project tasks. In software development methodologies, this task list is traditionally created using a top-down approach, in which

Conference'04, Month 1–2, 2004, City, State, Country.

tasks are broken into smaller more manageable subtasks. This approach strongly appeals to management because it seems orderly, predictable, and facilitates the allocation of resources [6]. While the top-down approach seeks to decrease task complexity, it increases project complexity because at some point all of the subtasks have to be integrated. Additionally, tasks may be prioritized and scheduled based on a risk or user-driven attributes like feature requests in iterative development cycles.

Once a task list is created, the tasks are organized into a task timeline in the form of a Gantt or Program Evaluation and Review Technique (PERT) chart. The Gantt chart, named after H.L. Gantt [9], is widely used to represent projected schedules and actual task progress against time. PERT charts allow teams to manage the interdependencies between tasks.

In Agile software development, tasks are identified using a bottom-up approach. Individual tasks are established and then built up to more complex solutions. Teams are empowered to decide what they will work on and how they will do it. The Agile method, eXtreme Programming (XP), takes the managing activity to the extreme (no pun intended) by specifically banning the use of PERT charts [14]. The philosophy holds that PERT charts are built on the faulty assumption that the tasks of a project can actually be positively identified, ordered, and reliably estimated. Teams using Agile methods may employ other techniques such as whiteboards and stickies to manage task coordination and execution.

3. LITERATURE SURVEY

We surveyed a number of case studies and ethnographies of software development teams to identify project management activities. The development teams ranged in size from as few as 5 to more than 120 developers. The application domains included enterprise Internet solutions, telecommunications software, medical solutions, configuration management tools, government information systems, and commercial single-use software. Teams in these studies were practicing both formal and informal software development methods. The projects employed a variety of development methodologies including: Rapid Application Development (RAD), open source software (OSS) development, traditional or waterfall development processes, and several *ad hoc,* unspecified or unknown methodologies. Collaborative planning activities include brainstorming project task lists, identifying task attributes, such as estimating deadlines, recording task interdependencies, and assigning roles and responsibilities, and reporting task/project status.

3.1 Project Management Breakdowns

Despite the existence of "best practices" and standard project management tools, breakdowns in the activities of managing still occur. Project team members can have misunderstandings about the development process and can become isolated from each other and the rest of the organization, the coordination process may become bogged down so as to incur a schedule delay, and the scheduling tools may not adequately provide for the needs of the project team. The following subsections highlight how current project management tools contribute to these breakdowns.

3.2 Tools Obfuscate the Process

Because project schedules organize information by tasks, a project team using these tools is forced "to organize work by task and not by person"[20]. This can make it difficult to get a handle on what other team members are doing and how one's own work fits into the whole process. Additionally, there is no way to link other management artifacts such as requirements documentation, test plans, or application programming interfaces (APIs) to the schedule. de Souza reports that because there was "no formal process to create and maintain APIs in the project plan" and therefore schedule, the APIs were forgotten until the last minute and caused additional schedule delays.[7]

Grinter similarly reports that Configuration Management tools did not create visibility into the development process. "At a higher level of abstraction, removed from the details of individual changes, the developers could not see how their work or other people's fitted together."[10]

Since project schedules contain a list of tasks and offer the ability to assign people to complete those tasks, it should be clear by looking at the schedule who does what. Yet misunderstandings about roles and responsibilities of project team members continue to crop up [4, 5]. This problem is exacerbated by the fact that team member roles and responsibilities can change over the course of the project [2]. Geographical distance and subtask complexity can also exacerbate the problem in large, distributed software development projects.

3.3 Schedule Can Isolate Team Members

In some cases the project manager can become isolated both from members of the project team and/or the rest of the organization. This can happen when the schedules and other work products created or maintained by project managers are incomplete or out-of-date. "Although most managers had developed progress tracking schemes, many were less aware of system status than were their system engineers."[4] How can it be that the person maintaining the schedule is so out of the loop?

Herbsleb found that reliance on documents can lead to impoverished and slow communication [13]. "There is a strong requirement for frequent updates of the schedule, so that it correctly reflects the current state of the project."[20] This can require daily maintenance without which the schedule can quickly become obsolete. Relying on these charts can cause project managers to overvalue the schedule to the process and consequently isolate themselves from the rest of the activities.

3.4 Tools Not Tied to Practice

Project management tools are not adequately tied to the practice of managing a project. Standard tools for maintaining project schedules are not linked to status reporting processes and require that the updates be funneled through a single person. In many cases, project teams reply on email status reports or 'today' messages to support group awareness [1, 12, 20]. This self-reported data may include as much or as little information as the developer wishes and as seen elsewhere, tight deadlines can encourage developers to be sparse with their comments [10]. When received by the project manager via email, the status report data must often be manually entered into the appropriate scheduling system to update the schedule. Consequently activities can feel like busy-work to the project manager and artifacts useless to the project team if the information is behind the times.

In some cases, status report and task data was conveyed via face-to-face meeting. Yet we have seen instances where "meeting data was not collected, so that information about decisions, rationales, and responsibilities was lost."[19] Teams also had difficulties assimilating new members to on-going projects because there was no connection between streams of communication [12]. Meeting data that is collected may be entered into word processing documents or spreadsheets so that it can be emailed to the project team. Again, there may be no ties connecting this data back to the project schedule.

3.5 Unintelligible Project Schedules

Geographically separated development teams are more likely to have different development philosophies and make use of different terminology. For example, the role of the project manager in one group may differ from that in another part of the organization, causing confusion about responsibilities [5]. The terminology used in the task list and project schedule can create a problem of mutual unintelligibility. "For the schedule to be interpretable by all, it must use a shared vocabulary."[20]

"Large groups of 'project size', as Grudin (1994) calls them, can not find out what the status of the project is by social interaction alone."[4] As project team size grows, the number of tasks grows, and Gantt and PERT chart representations make it easy to get lost in the schedule. Large groups relying on the project schedule for status information may require matrix printers and entire walls to display such huge amounts of data. Such large schedules can make even simple navigation of the schedule problematic.

4. DISCUSSION

4.1 The Social Aspects of Managing a Project

In large-scale development teams, membership in teams and work on projects is not static. Team members are likely to be working on more than one project at a time. Additionally, project team structures can change to meet the needs of the organization. The assimilation of new project members becomes a project management activity. Project management tools can aid the assimilation of new team members by providing a contextualized knowledge record of the project.

The collaborative generation of project tasks and task attributes is a social activity and can produce a greater mutual understanding and expectations for the project. Collaborative scheduling empowers team members to determine task deadlines and may improve task deadline estimates. Additionally, making a schedule visible in a social context influences how the schedule is understood and interpreted by project members. Whittaker notes that publicly displayed project schedules support collaborative reflection on the project and task deliverables.[20] In comparing a publicly displayed wall schedule vs. an electronic version, "the board was considered more 'real' and 'credible' than traditional electronic schedules".[20]

4.2 Role of Documented Schedules

The use of formal documents impacts the flow of communication in organizations in many different ways. Charts and graphs can improve the vertical flow of information, because they abstract data to a level which informs management about the progress of activities. [22] Email and other informal written communication can improve the horizontal flow of information. [1, 12] Many development teams use email as a primary method of communication amongst developers quite successfully. Subtask groupings in schedules and APIs can serve to reify organization boundaries, which may impede the flow of information [7]. Having a clear understanding of the role of project schedules in team communication is a necessity to future tool development.

In summary, project management tools, namely the project task list and project schedule have not changed much since their introduction. Because of the high-level of goal uncertainty in software development these tools may not be meeting the needs of project teams. Software tools for creating project schedules were initially developed as single-user software, and despite new collaborative features continue to be used as such. Unlike standard MS Office offerings, MS Project may not be installed on everyone's computer and so the outputs from the tool such as a Gantt chart are no more interactive than a paper printout. The source data may not be publicly available and therefore difficult to keep current. Scheduling tools give little guidance to support a task structure that is universally understandable. The current state of the project schedule can obfuscate the development process instead of making that process visible to the development team. Finally, these tools are designed to be used in a myriad of contexts so much so that they may be too generic for contextualized use.

5. FUTURE DIRECTIONS

Research contends that there is no one-size-fits-all methodology [3], so why are we using one-size-fits-all tools? Perhaps its time to move away from the PERT chart view of project management. Software development projects are known to be more uncertain than other types. A few methodologies have strived to overcome uncertainty issues with project management tools by avoiding them altogether. In XP, for instance, no formal project schedule is created. In these methodologies informal communication is expected to suffice for managing activities. However informal communication will not be sufficient in large-scale, distributed software development projects.

Based on the data drawn from current project management practices in software development teams, we make the following recommendations in the future direction of project management tools:

- □ Make the project management process more visible.
- □ Align tools with planning practices.
- □ Link tools to other project artifacts.
- □ Strive for a people-oriented, not task-oriented focus.
- □ Tool should be accessible by all to promote individual responsibility and collaborative planning.
- □ Tools should be publicly viewable to promote informal communication, collaborative reflection.
- □ Make the tools easy to keep update and maintain.
- □ Try other visualizations for identifying and enumerating tasks and task attributes.

6. CONCLUSION

In this paper we have discussed the social-aspects of project management in software development. Based on an analysis of a survey of the literature, we have made some recommendations for the future development of collaborative project management tools. We hope that future development may be inspired by other visualization techniques such as social network maps and activity rhythms.

7. ACKNOWLEDGMENTS

We would like to acknowledge support by Intel Corporation for this research. We thank our colleagues at the UW iSchool for their input and reflection.

8. REFERENCES

[1] Bernheim-Brush, A. J., & Borning, A. (2003). 'Today' messages: lightweight group awareness via email. In CHI '03 extended abstracts on Human factors in computing systems (pp. 920-921). Ft. Lauderdale, Florida, USA: ACM Press.

[2] Beynon-Davies, P., Mackay, H., & Tudhope, D. (2000). It's lots of bits of paper and ticks and post-it notes and things…': A Case Study of a Rapid Application Development Project. Journal of Information Systems, 10(3), 195-216.

[3] Charvat, J. (2003). Project management methodologies: selecting, implementing, and supporting methodologies and processes for projects. Hoboken, NJ: John Wiley.

[4] Curtis, B., Krasner, H., & Iscoe, N. (1988). A Field-Study of the Software-Design Process for Large Systems. Communications of the ACM, 31(11), 1268-1287.

[5] Damian, D.E. & Zowghi D. (2003) RE Challenges in Multi-site Software Development Organizations. Requirements Engineering 8:149-160.

[6] DeGrace, P. & Stahl, L.H. (1990) Wicked Problems, Righteous Solutions. Englewood Cliffs, NJ. Yourdon Press.

[7] de Souza, C. R. B. d., Redmiles, D., Cheng, L.-T., Millen, D., & Patterson, J. (2004). Sometimes you need to see through walls: a field study of application programming interfaces In Proceedings of the 2004 ACM conference on Computer supported cooperative work (pp. 63-71). Chicago, Illinois, USA ACM Press.

[8] Fox, T.L., & Spence, J.W. (1998) Tools Of The Trade: a survey of project management tools. Project Management Journal 29(3), 20-27.

[9] Gantt, H.L. (1919) Organizing For Work. Harcourt, Brace, & Howe. New York, NY.

[10] Grinter, R. E. (1995). Using a configuration management tool to coordinate software development. Paper presented at the Proceedings of conference on Organizational computing systems, Milpitas, California, United States.

[11] Grudin, J. (2006) Enterprise Knowledge Management and Emerging Technologies. *Proceedings of the 39th Hawaii International Conference on System Sciences* (HICSS-39). 10 pages

[12] Gutwin, C., Penner, R., & Schneider, K. (2004). Group awareness in distributed software development. Paper presented at the Proceedings of the 2004 ACM conference on Computer supported cooperative work, Chicago, Illinois, USA.

[13] Herbsleb, J.D. & Mockus, A. (2003) An Empirical Study of Speed and Communication in Globally-Distributed Software Development. IEEE Transactions on Software Engineering 29(6), June 2003.

[14] Larman, C. (2003) Agile & Iterative Development: a manager's guide. Boston, MA. Addison-Wesley.

[15] Lin, J., Newman, M.W., Hong, J.I., Landay, J.A. (2000) DENIM: finding a tighter fit between tools and practice for web site design. In the Proceedings of CHI 2000. April 1-6, 2000. The Hague, Amsterdam.

[16] Maylor, H. (2001) Beyond the Gantt Chart: project management moving on. European Management Journal 19(1), 92-100.

[17] PMI. (2006). Project Management Best Practices (PMBOK). Retrieved March 3, 2006, 2005, from http://www.pmi.org/prod/groups/public/documents/info/pp_pmbokguidethirdexcerpts.pdf

[18] Sorensen, R. (1995, January 1995). A Comparison of Software Development Methodologies. Crosstalk, 1, from http://www.stsc.hill.af.mil/crosstalk/1995/01/comparis.asp

[19] Walz, D. B., Elam, J. J., & Curtis, B. (1993). Inside a Software-Design Team - Knowledge Acquisition, Sharing, and Integration. Communications of the ACM, 36(10), 63-77.

[20] Whittaker, S., & Schwarz, H. (1999). Meetings of the Board: The Impact of Scheduling Medium on Long Term Group Coordination in Software Development Computer Supported Coop. Work 8 (3), 175-205

[21] Yates, J. (1993). Genres of Internal Communication. In Control Through Communication: the rise of system in American management (pp. 65-100). Baltimore: Johns Hopkins University Press.

Social Dependencies and Contrasts in Software Engineering Practice

Jonathan Sillito
Department of Computer Science
University of Calgary
Calgary, AB Canada
sillito@cpsc.ucalgary.ca

Eleanor Wynn
Strategic Capabilities, Information Technology
Intel Corporation
Hillsboro, OR U.S.A.
eleanor.wynn@intel.com

ABSTRACT

This paper reports initial observations from a qualitative study of software engineering at a large technology company. Data were collected from interviews with software engineers and managers, formal company documents, and observations of group and team meetings. This is an early assessment of analytical categories we believe are important for understanding informal work processes and flow in a distributed software engineering team. These categories include the distributed nature of the organization; ownership and dependencies between code modules; and process improvement initiatives. Findings suggest that software engineers think about module dependencies as people dependencies; and reveal contradictions between the motivations of diverse and often concurrent process improvement efforts.

1. INTRODUCTION

We report on initial observations from a qualitative study of software engineering at a large technology company. Our observations are based on interviews with software engineers at the company, company documents and team meetings. Specifically, data for this study came from approximately thirty-five interviews of both managers and software engineers at the company. The interviews mainly took the form of conversations, and lasted between fifteen and thirty minutes. We regularly attended group and team meetings and read company documents relating to software engineering processes and issues. Much of the information we present in this paper is in the words of our participants or is based on quotes from documents we have reviewed. For each quote we identify the source as a software engineer, a manager or a document. The quotes reflect perspectives of the participants. All interviews and meetings took place during the summer and fall of 2005.

Our analysis focuses on social aspects of software process, as possible keys to fully understanding software engineering at this company. Our thinking follows the lines established by Dalbohm and Mathiassen [6] and Cibora [4] about the importance of considering social issues in the study of technical systems. Data analysis is preliminary; and our aim is to provide a relatively low-inference view of our data. Our reason for presenting results at this early stage is that we believe insights will inform other researchers working in social aspects of software engineering. We also hope to generate discussion and get feedback to direct further analysis and research directions.

In the next section we briefly describe aspects of the organizational setting of our study (see Section 2). Then we present our observations about source code ownership and dependencies (see Section 3); followed by observations about attempted process changes, including efforts based on CMMI, extreme programming, and the open source model. We conclude the paper with a summary of key themes (see Section 5).

2. THE SETTING

We studied software engineering at a large technology company which has thousands of employees working on software development. In this paper, we refer to these employees as software engineers (SEs). They are located in multiple divisions within the company hierarchy and are globally distributed. Many software engineers in the study group focused on providing support for very large hardware engineering groups working on time-critical, high-value designs. Some SEs spend almost all of their time on software development. Others split software development with other responsibilities such as operations or architecture (for example, one software engineer we spoke with spent one third of his time coding and the rest on operational support). The software written and maintained by the software engineers we met with ranges from *"small projects"* and *"one off scripts"* to multi-million line programs, including both internal and external applications.

The company is geographically distributed with software engineering sites around the entire globe. Several of the groups (employees who report to the same direct manager) and teams (employees working on the same project, possibly reporting to different managers) samples are distributed between two or more countries as well as different states within the US. This includes countries that are on essentially opposite sides of the world. One team had members in Israel, Russia and two different US states. Another group was distributed between the US and India.

When groups or teams are not collocated, meetings are held using phones and a multipoint video conferencing application which supports sharing desktops. Many employees expressed a preference to meet in person: *"we are getting so distributed that when I get a chance to meet some one face-to-face it is so exciting"* [manager]; and *"I am better face-to-face"* [SE], but other research shows that employees can cope well with distance even when it is not preferred, depending on task complexity (e.g., [9]). Some face-to-face interaction was believed to be important for trust and aspects of collaboration: *"some people need face time to build trust"* [manager]; and *"face-to-face collaboration is essential for effectiveness"* [manager]. Some travel occurs, frequently at the beginning of projects, to provide this social linkage (*"the project began with some intense face-to-face meetings in various locations"* [SE]); one long standing team meets weekly by phone and yearly in a *"face-to-face"* [SE].

When teams are widely spread across time zones, for one or more participants, meetings end up occurring at inconvenient times (see for example [8]): *"time-zones are awkward, we often have 7am meetings"* [SE]. Rotating the time for meetings to vary the team member who is inconvenienced was commonly referred to as *"sharing the pain"* [SE]. Several managers expressed challenges for their groups in working across time-zones: *"time zone makes it hard to trouble shoot and interact with customers"* [manager]; *"I find that working across time-zones in this way results in lots of misunderstandings and rehashing—you think you are in agreement but then later realize you are not"* [manager]; and *"major collaboration problem is time-zone—this problem is structural, unchangeable"* [manager].

The employees we spoke with, expressed concerns about collaboration or coordination that may be normal for employees at a large, highly distributed company: *"doesn't seem to be much cross pollination anywhere"* [SE]; *"we waste time solving a problem already solved elsewhere"* [document]; and *"lots of reinventing the wheel in our department"* [manager]. One aspect of collaboration issues were concerns raised over community, including accessing or making available expertise on a particular software development technology:

> *"There could be 50 different people using this [technology], but I don't know. We are the experts [. . .] but everyone is inundated with email and newsletters. We just scan them. So how do you get the word out there?"* [manager].

In addition to the company size and its geographical distribution, concerns about collaboration may also be related to the company culture around information, including source code and other software project information, ownership (discussed further in the next section) and the sharing of that information, which the author of one document felt was heavily based on *"tribal knowledge"* [document]. A manager felt that the state of development tools was also part of the issue: *"[current] collaboration [tools] are not very rich for developers"* [manager].

3. OWNERSHIP AND DEPENDENCIES

There are different source code ownership models in the company, though generally ownership falls within organizational groups. Access to and awareness of that code is also often limited along organizational lines. In some groups, smaller code bases or modules are owned by individuals and the associated support structure made some software engineers hesitant to take on certain commitments or to release code beyond their local context, lest they be required to *"support it for life"* [SE]. One group we observed was moving from this model to one of *"shared ownership of code"* [manager]. We also observed that some groups own (or are organized around) a particular software development technology area: *"so in all those four layers we have expertise within our group"* [SE].

Software engineers from two different groups said that their code bases are split into multiple versions *"because it's from different organizations"* [SE]. In other words, in some cases, different groups own different forks (or branches) of a the same code base. One software engineer called this approach to handling forks a *"recipe for disaster"*. In these cases a common task (*"I do this lots"* [SE]) is to *"replicate the same work by separate people because they are different groups"* [SE]. Describing his copy and paste approach to avoiding this duplication of effort one software engineer said *"so being the lazy engineer that I am I went and got their work and brought it over. I am sure that it's probably not going to work first time around, but it will be a lot easier to debug this than start from scratch"* [SE].

At times software engineers expressed dependencies between code modules in terms of dependencies between people or between organizations; and at times software engineers talked about locations in the world or the building when we would expect them to refer to locations in a code base: *"all the code in the department"* [SE]; *"that module is provided by another guy downstairs"* [SE]; and *"[my] module that pretty much the rest of the department is going to depend on"* [SE]. One software engineer duplicated code to manage or avoid a dependency on another person: *"so I ended up taking a lot of code from other places and bring them in here"* [SE].

In one case, where several groups each had ownership for different large modules that were all part of a very large (more than 1 million lines of code) and critical application, effort was put into isolating those modules by minimizing the dependencies between them so that software engineers *"can get away with just knowing [their] little bit"* [SE]. One of our participants said, of this effort: *"however all the complexity, well big chunks of the complexity get moved into the translating from one data model in to the next"* [SE]. Another member of that same group said: *"what happens is a lot of times in integration, when you integrate that module into the bigger system, something goes wrong"* [SE]. Further, access to code may be organized in a similar way: *"if I wanted to grep through all of the code in the department I wouldn't be able to—I don't know how to get to all of the code"* [SE]. This finding is also reflected in a report by de Souza et al. claiming that the use of APIs impacted forms of collaboration between groups [7].

Making changes to production code is naturally difficult and members of some groups are understandably cautious, partly because any changes *"might break other code"* [SE]. One software engineer felt that *"unit tests cannot be written for all types of code [. . .] simply because you see unit tests attached to code does not mean that the code is actually being sufficiently tested"* [SE], which creates a confidence problem when changes are made. Examples of software engineers expressing this hesitation to make changes include: *"this could use some improving but everything is tied into this, so no one wants to mess with it"* [SE]; and *"when refactoring, what we do is in order to not break things for other developers, we leave the old code in place"* [SE]. One group has a formal change *"process"* [SE] which involves multiple meetings between various parties, with the consequence that *"when we change a core component it usually takes a long time, even for small changes, and for the most part we just don't change them"* [SE].

Like dependencies between modules, issues around making changes are expressed in terms of relationships between people: *"I think the hardest parts are [three certain modules] because you have so many people depending on your code. . . very difficult to know what changes are safe, and so there is a natural reluctance to make changes"* [SE]. Dealing with situations like this naturally becomes a coordination problem between people and groups (*"organizationally you have a person on [team A] who is just assigned to talk to [team B] and then on [team B] you have someone who is assigned to talk to [team A]"* [SE]; and *"lots of negotiation: a back-end folk and somebody from our side"* [SE]) rather than an issue primarily handled using code exploration or debugging tools.

4. PROCESSES AND PRACTICES

During our study, we observed three types of process improvement efforts: Capability Maturity Model Integration (CMMI) based efforts [3], agile or extreme programming based efforts [5, 2], and efforts inspired by the open source model [10]. These efforts varied in their motivations, in who initiated them (i.e., management or the software engineers themselves) and the organizational scale of the effort.

The CMMI based process improvement efforts were large scale, official efforts, and were typically initiated by high-level management: *"CMMI is the way we manage work, we have level two and are moving forward with key process areas"* [manager]. The primary motivation for was to improve control and consistency: *"the impetus was that we were an organization of 3500 people with software development as the core competency, but we couldn't tell how things were being done. We needed a consistent process to fix the issues"* [manager]. In addition to control and consistency, CMMI was intended to improve timeliness, quality, and reduce rework and defects in released code. The initial goal of one organization was to be at level three in eighteen months, but *"two and a half years later we are not even close."* [manager]. For this organization reaching a particular certification level is no longer the focus, the focus is now on solving specific problems with their software engineering processes.

Employee reaction to the CMMI initiative was mixed. Some employees were openly skeptical: *"customers don't care about levels... I have seen a company at level four that produces garbage. CMMI should be approached as a problem solving exercise, not for its own end"* [SE]. Others felt the particular approach to CMMI taken by their organization was flawed: *"what is in place is too rigid or too engineered"* [manager]; and *"I would like a much simpler interpretation"* [SE]. We also observed a more general interest in *"simplifying and streaming our software development procedures"* [SE]; and CMMI initiatives were seen simply as overhead by some software engineers: *"I would like to have more time actually doing what I consider to be the meat of my job, which is development, including design and analysis and all that stuff and less time messing with the bureaucracy of the company"* [SE]. While some groups or teams aimed to put in a minimal amount of effort to satisfy the requirements (*"the way we do it is, tell us what we need to do satisfy our CMMI requirements"* [SE]), others were more enthusiastic (*"our team is kind of zealous about CMMI and we really don't like dragging our feet. . . Our team is really unique in that we really do a lot of the CMMI based things that aren't necessarily on [the company] road map. We do validation, verification of every work product. . . and we do a traceability matrix that we got from and old CMMI thing. We are trying to do as much as we can from beginning to end. . . There are still gaps but we are trying the best we can. Especially with the things that we can see will add value right away, which most of it does"* [SE]).

Efforts to follow an agile approach originated at the group level, often having been initiated by the software engineers themselves. This was less systematic or widespread than the CMMI efforts but appeared to be gaining momentum: *"there seems to be more of a trend of extreme programming picking up within [this company]"* [SE]. The primary motivation for agile approaches generally seemed to be to improve responsiveness and speed. For example, one manager described his group's motivations for using extreme programming: *"We have been using agile methods for about two years. The push came from our customers who wanted things faster... We succeeded, but we are not as fast as we hoped"* [manager].

The groups made adaptations to extreme programming practices to fit their needs. Most of these were done expressly to accommodate the distributed nature of the work environment. For example, stand-up meetings took place using a phone bridge (because the team was *"tending to be more distributed"* [SE]). One group was starting to use more upfront design to reduce the coupling within the group: *"two parts of our group [one in India and the other in the US] working together is causing work life balance problems"* [manager].

Pair programming seemed to be the most often modified extreme programming practice. One pair we observed used a screen sharing application to share their desktops with each other (though they were collocated) and there was no clear "driver" and "observer" roles [11]. Another pair (distributed between the US and Israel) shared responsibility for the same aspects of a code base but did not program together:

> *"When I say pair programming I guess I mean doing the design together, doing peer review of the code, maybe the tests will be written by the other person, but not so much in the extreme programming sense of sitting down together"* [SE].

Several groups we met with have used screen sharing for pair programming between geographically distributed pairs. This approach was not viewed as very successful: *"pair programming over [screen sharing]is very tiring and not very enjoyable..."* [manager]; *"one other real-life problem we have encounter is that pairing does not naturally allow developers to take time to really think a problem through if one is unexpectedly encountered [. . .] This is especially problematic for virtual pairs, since it appears awkward for developers to be on the phone in silence for extended periods of time, sometimes even just for a few seconds. So the developers just keep on talking, digging themselves deeper into the problem, rather than calling for a time out"* [document describing one teams experience with extreme programming].

One software engineer expressed his opinions on the CMMI initiative and the extreme programming efforts in his group this way:

> *"CMMI is very rigid. XP can be very loose depending on how you interpret it. We feel in this group that we need some middle ground between the two. So we are not quite happy with either process. That's what*

> *it boils down to. I think XP went a little overboard."* [SE].

Though the general attitude towards extreme programming was positive (*"experimented with XP... the results of which were basically positive. The biggest win was test driven development"* [SE]; *"XP is way better than the other models I have followed. I love XP... everyone knows where you are"* [SE]), one software engineer made this comment about extreme programming as performed by a team he was not on: *"XP methodology just doesn't deliver: [a certain project] has gone on for three years and has still not delivered. For most of it they have been close"* [SE]. Interestingly, a member of the team he has referring to said that *"XP is working very well for us; we are more of a team"* [SE].

Process improvement efforts based on the open source model were on a smaller scale, though there was some official interest in the organization, including experiments involving making various supporting tools available. These efforts appeared to be motivated by a desire to avoid duplication (*"too many point solutions"* [SE]), to resolve code ownership and support issues such as those discussed above, and generally to improve the level of collaboration, both within groups and between groups. Open source is viewed as more community oriented, and by nature, non-proprietary, so it would allow for group standardization rather than externally imposed standards on software. Our opportunity to investigate the open source initiative was limited and may have occurred more extensively in other groups than those we observed.

5. SUMMARY

The results we have presented reflect preliminary observations; further analysis and data collection will undoubtedly deepen the view into the exploration of ownership and the tensions around process. Historical developments may also overtake or provide nuance to what we describe above. Still we believe the results expose valuable leads to themes and issues from within team experience as collected by a participant observer with steady access to the environment. We summarize these themes along with possible future research directions below.

The first theme concerns ownership and sharing of code and other artifacts. Software engineers often expressed dependencies between code modules as dependencies between people and groups. Dealing with code integration is an organizational rather than purely technical matter. Many software engineers would clearly prefer not to make changes to production code because of the overhead imposed by change processes (including change committees), between-group friction, and uncertainty about testing. One way to pursue these issues further would be to look deeper into ownership models and approaches to version management, and to study organizational issues in change management.

Process improvement efforts remain an area of interest for ongoing study. CMMI initiatives are largely seen as a method to improve consistency. Extreme programming initiatives are efforts to improve responsiveness and flexibility. The extreme programming practices that appear to work poorly in a distributed environment include pair programming, lack of upfront design and standup meetings (which end up being at the end of the day for some people). Process improvement efforts inspired by the open source model are seen as solutions to issues around ownership and duplication of effort. Going forward, a relevant research issue will be how well the various improvement efforts address expected challenges and what factors (at various levels) differentiate successful initiatives from unsuccessful ones.

CMMI based initiatives were typically at a large scale and initiated by high-level management. Agile based approaches were typically local efforts initiated by the software engineers in a particular group. These diverse process improvement efforts are likely to come more and more into direct contact and contradiction. For example, a team that has decided to pursue an extreme programming approach will be instructed to begin following a prescribed CMMI based initiative. Scenarios such as this raise important questions about bottom-up versus top-down improvement efforts and how well these efforts can coexist, commingle, reconcile or be integrated. Further research could consider these contrasts in the context of existing theories (from the military domain) that propose the integration of command and control with tactical freedom [1].

6. REFERENCES

[1] David S. Alberts and Richard E. Hayes. Power to the edge. *US DOD Command and Control Research Center Publications*, 2003.

[2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, 2004.

[3] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. Addison-Wesley Professional, 1st edition, 2003.

[4] Claudio Ciborra. *From Control to Drift: The Dynamics of Corporate Information Infrastructures*. Oxford University Press, 2000.

[5] Alistair Cockburn. *Agile Software Development*. Addison-Wesley Professional, 1st edition, 2001.

[6] Bo Dahlbom and Lars Mathiassen. *Computers in Context The Philosophy and Practice of System Design*. Blackwell, 1993.

[7] Cleidson de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: A field study of application programming interfaces. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 63–71, 2004.

[8] Alberto Espinosa and Cynthia Pickering. The effect of time separation on coordination processes and outcomes: A case study. In *Proceedings of the International Conference on System Sciences*, 2006.

[9] Cynthia Pickering and Eleanor Wynn. An architecture and business process framework for global team collaboration. *Intel Technology Journal*, 8(4), 2004.

[10] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 1st edition, 1999.

[11] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair-programming. *IEEE Software*, 17(4):19–25, 2000.

Architecture to Support Team Awareness in Large-Scale Software Development

Andrew Sutherland, Tadeusz Stach, Kevin Schneider, Carl Gutwin
University of Saskatchewan
Saskatoon, SK, Canada
1-306-966-4886
{andrew.sutherland, tad.stach, kevin.schneider, carl.gutwin} @usask.ca

ABSTRACT

Developers in large-scale software development projects depend on multiple sources of information in order to stay aware of the activities of other team members. For example, a programmer will attend meetings, engage in conversations with other developers, read mailing lists, and examine version control logs so that duplication of work and conflicting changes are avoided. Gathering and tracking this information requires additional effort, and adds to the already complex process of writing software. In this paper, we describe a technique that allows developers to dynamically receive awareness information as they develop, maintain, and document software. The technique allows for collecting diverse awareness information from multiple sources; increasing the likelihood that important awareness information is available to the user.

Categories and Subject Descriptors

D.2.10 [Methodologies] – *group awareness, design, large-scale software development*

General Terms

Human Factors, Management, Measurement

Keywords

Team-oriented development, tool support, architecture

1. INTRODUCTION

Group awareness in software development refers to having knowledge of other developers' activities as they make changes to the software. This knowledge is necessary to avoid duplication of effort and conflicting changes when performing software development tasks.

A study of open-source development teams [3] showed that open source developers maintain group awareness primarily through textual channels such as email lists, text chat, and version control system logs. This same study also identified a desire and need for tools that can assist in gathering, filtering, and interpreting the information obtained from awareness information resources.

Our approach integrates *awareness entities* with software project artifacts. Awareness entities link and distill the various sources of awareness information to allow a software developer to maintain team awareness while performing a number of software engineering tasks. For example, with awareness entities it is possible to seamlessly navigate information about a team's activities while navigating the project's source code. The developer does not need to search through volumes of email, news groups, project plans or change logs. Our intent is to explicitly represent awareness entities in the architecture of the software system.

1.1 Motivation

A number of tools have been developed to support awareness in team oriented software development [1][2][4]. Although these tools have been shown to be effective at visualizing or expressing certain awareness data, they have not been adopted for general use by development teams. We propose there are three reasons for this lack of enthusiasm.

The first reason is that awareness data is maintained using a multitude of information sources, including (but not limited to) source code repositories, e-mail lists, chat histories, and bug tracking programs. Existing tools have primarily focused on mining information from only one or two of these sources. This means the provided awareness information is often incomplete, as it has been shown that developers often rely on most or all of these information sources [3]. For example, it has been demonstrated in some systems [2][4][7] that the facts from the CVS repository of a project can be mined to inform the user of the software artifacts that have been worked on and by whom. Although this is useful information, it can provide a false sense of the expertise and responsibilities among the developers. For example, a person may have performed many CVS commits on a system file; however this person is actually a junior developer responsible for committing files once the senior developers have completed their code changes. If a user were shown the number of e-mail conversations between the senior developers regarding the particular file, perhaps a better understanding of the roles and responsibilities would be obtained.

The second reason for lack of use of awareness tools is that they often focus on the gathering of the data itself, and do not provide mechanisms for the user to find the desired information. The data is often displayed in large quantities, and requires the user to search through the results in order to find what is usually a relatively small item of information. This issue can be addressed by making the returned results more specific to what the user is actually interested in finding out.

Finally, the volatile nature of awareness information also presents challenges. Changes to artifacts, and conversations about artifacts are constantly occurring within a large project. Supplying the user with up-to-date information becomes difficult due to the fast pace with which decisions and changes are made on large software projects.

To address these issues, we propose a new design for awareness support tools. This design is based on an architecture that simplifies the process of gathering up-to-date awareness information from a variety of resources. These resources are monitored by awareness entities that are responsible for certain artifacts in the software being developed. The information contained in these entities can include: a history of version control operations performed on the artifact (i.e., the number of times it has been checked in, checked out, or committed with other files), who has most recently handled the artifact, the related messages found in the mailing lists and chat systems regarding the artifact, and any reported bugs related to the artifact. These sources of information are monitored so that when new commits, messages, or other actions are performed, the information in the awareness entities is immediately updated and the developer can be supplied with the most up-to-date information.

Furthermore, since these entities contain awareness information only, they can be attached to a variety of presentation formats. For example, the entity could either be presented in a graphical visualization, or simply serialized to a text file for archiving.

2. BACKGROUND

2.1 Collaboration in Development

The collaboration among developers in large-scale projects has been an area of interest among both software engineering and CSCW communities. One area of research examines how programmers organize themselves and maintain awareness of a project (both distributed and onsite). Herbsleb and Grinter [5] examined how developers coordinate and communicate. Their research shows that dependencies are sometimes missed and that work is often duplicated or adverse changes are made to the code. These problems arise due to a lack of awareness about what is occurring in the project. Similarly, Gutwin et al. [3] looked at how developers maintained awareness in several open-source projects. Their findings show that mailing lists and chat tools were most often used as a means of understanding what was occurring within the software project.

These studies reveal the need for tools to increase the awareness among developers. The software design proposed in this paper is meant to address the issues of collaboration in development projects by making it easier to maintain awareness of other's activities.

2.2 Tool Support for Awareness

Collaborative software development involves numerous actions and tasks: design, coding, documentation, version control, as well as online and offline discussion. All of these items provide information to the developer for a better understanding of a project. Several software visualization tools have been proposed in order to make it easier for developers to maintain an understanding of software projects. De Souza et al. [7] introduced the Augur tool for software visualization of large systems. Augur mines data from a CVS repository and encodes it visually. The tool is able to present several data elements within one visual frame. For example, line type (i.e. method, comment, class, etc), authorship, and the date of last modification for each line of code in a project can all be revealed in a single view.

Gutwin et al. [4] developed a tool to increase awareness within a distributed development environment. The intent of the tool is to provide developers with easier access to awareness information. The awareness information is obtained from data mining the CVS repository used for a particular project and then visually displaying information such as who worked on what file, and what changes have occurred in the project. Although both Augur and ProjectWatcher can provide insight into particularities of a software project, the information is collected solely from a single source (e.g., a CVS repository). However, as Gutwin et al. [3] discovered, developers in distributed projects rely on several sources of information to understand the current state of affairs.

Research into large scale-scale software development has shown that mailing lists can be an important source of information [3] [5]. Despite this fact, there have been few tools which present this information to developers in a usable fashion, and they are therefore forced to search for items of interest manually. Viégas et al. [8] performed research on mining email archives for relationship data between correspondents, and presenting the results visually. This area of research has yet to be applied to software development projects, but has the potential to improve awareness among developers.

Cubranic et al. [1] presented the Hipikat tool, which provides recommendations to developers about which artifacts should be examined prior to making a change to the software project. Recommendations are based on the history of the software project. A unique feature of this tool is that data is mined from multiple sources: email lists, documentation, change reports, and CVS logs. Although Hipikat does not necessarily aide in group awareness, this is an important feature since it demonstrates that it is possible to gather and summarize data from multiple project artifacts.

Our design allows current information, collected from multiple data sources, to be provided to the developer in a variety of forms. This will allow for the creation of tools with the ability to gather data from several sources (e.g., CVS and mailing lists) with the flexibility to present the information in multiple formats.

A similar idea has been proposed by Kirsh-Pinheiro et al. [6]. They designed a framework for supporting awareness of certain events within groupware applications. Their approach differs from our approach in that we focus on providing the user with awareness of other developers' activities on a project, and our design is not limited to synchronous distributed groupware applications.

3. PROPOSED SOLUTION

3.1 Architecture Description

To approach the problem of supplying the user with relevant, up-to-date awareness information, we have based our design on an architecture consisting of a repository of awareness facts, and a series of *awareness entities* that are responsible for consolidating the facts related to a particular software artifact. Figure 1 shows a typical setup for our architecture. In this case, the awareness fact repository monitors the four awareness resources on the right.

The *Awareness Fact Repository* (AFR) is responsible for storing facts based on awareness information obtained from the awareness resources. The AFR monitors these resources and maintains a list of *awareness facts*. These facts are generated whenever *awareness events* occur in the awareness resources.

For example, if a system file *ViewManager.java* is committed to the CVS repository (an awareness event), a CVS log entry is generated. The AFR monitors the CVS commit log, and stores the following fact based on information present in the log:

```
Artifact: cs/discussion/ViewManager.java
Edited At: 2005-09-28 15:17:20
FactType: CVScommit
----------------------------------------
-
Total Edits:  2
Edited By: rss050
Revision: r3390
Comment: added 'synchronized' to all of
the getInstance() methods
```

This fact represents a particular type of awareness data – a CVS commit. The fact is indexed by the artifact name, time the fact was created, and the fact type. The fact type corresponds to the resource from which the fact was generated. This means that the *ViewManager.java* artifact can have facts of multiple types associated with it, each corresponding to a different time and/or awareness resource.

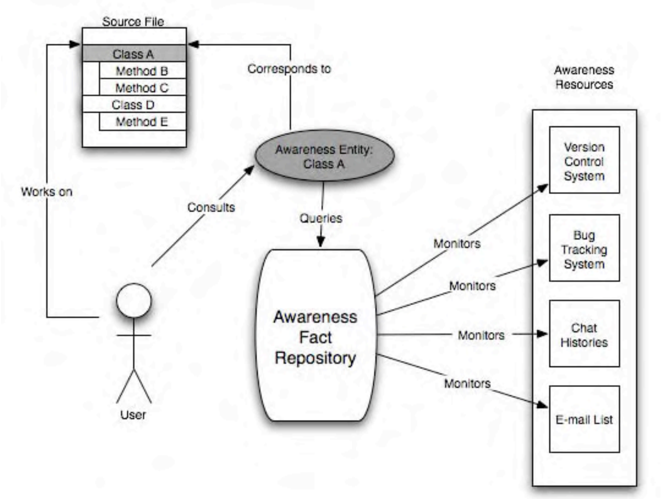


Figure 1 – Illustration of an architecture for supplying dynamic specific awareness data

In addition, the *ViewManager.java* artifact could have awareness facts generated from bug reports, e-mail messages, chat histories, or any other source of awareness information. Also, the design is extensible, and so monitors for new types of awareness resources can be added. Additional functionality can also be added to the AFR for monitoring a different resource type. This and other implementation details we be discussed further in the next section.

Since new facts are generated whenever an event occurs in an awareness resource (e.g. CVS commit, new e-mail, bug report, etc.), the AFR always has the most recent awareness information stored. The other advantage of storing the awareness data in this fashion is that it becomes a simple task to quickly retrieve awareness data pertaining to a particular artifact. The awareness information is keyed to the artifact name, date, and resource it was obtained from. This means that if data pertaining to a certain artifact in a particular period of time is desired (e.g. who changed *ViewManager.java* in the previous week) it is a simple matter to retrieve the facts that meet these criteria. This allows for the creation of *awareness entities*, which are objects consisting of consolidated awareness facts. The awareness facts are consolidated according to what software artifact they are relevant to. For example, the shaded awareness entity seen in Figure 1 contains awareness facts concerning *Class A*.

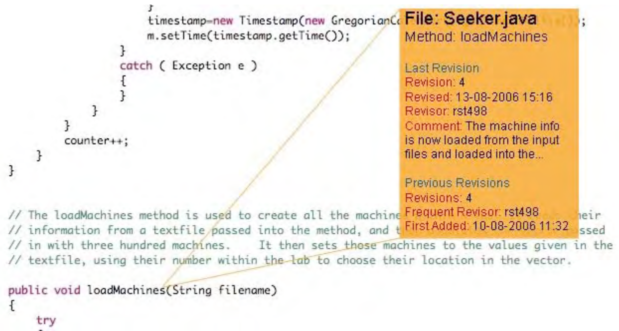
Consolidation of the relevant awareness facts into an entity has a number of benefits. The awareness entity can be treated like another software artifact. For example, it can be represented in a visualization application as an entity that can be navigated to, associated with other artifacts, and stored for later use. It also allows multiple members of a team to refer to the same group of awareness data, which may prove useful in supporting the social aspect of software development. We also believe that giving awareness data a more substantial representation in the architecture will open avenues for future research that will allow us to study navigation of awareness information, just as the navigation of regular software artifacts is studied.

3.2 Implementation

To implement monitoring of the awareness resources, the AFR will compare the latest version of each awareness resource it is monitoring with the current version at certain time intervals. Since these awareness resources are based on textual information, it is a simple matter to detect changes in the content. Differences between the previous version of the text and the recent versions will cause the AFR to generate new facts based on what was changed. More often than not, the changes will simply be additions to the textual awareness information (e.g. a log entry for a commit or checkout, a e-mail sent to the group, etc.).

The consolidated awareness information used to form the awareness entities is gathered from the AFR. The awareness entity is instantiated when the user triggers the system, informing that awareness information concerning a certain artifact is required. How this triggering of the system occurs depends on the nature of the system being used to develop the artifact. For example, it may be that the user begins editing a certain source code file, or informs the system that he wants to know who last updated certain design documents. It could also be that the user selected a particular artifact while using a visualization application.

The manner in which the awareness entity is presented to or interacted with by the user is flexible. A system that utilizes this architecture may provide the user with direct access to the textual information stored inside the entity through a pop-up information box as in Figure 2. Alternatively, it may be useful to interpret the entity by presenting it graphically. The awareness entity could be mapped to a graphical representation, in order to perceive qualities about the data and its relation to the software. Figure 3 demonstrates how an awareness entity containing facts about developer activity on a certain software artifact can be visualized. In this case, the length of each bar represents how much each developer contributed to the coupling of the corresponding artifact in relation to another artifact. Note that if this information had not been readily available in an awareness entity, the visualization application would have to mine the entire CVS repository for facts pertaining to this artifact.



```
                  }
                  timestamp=new Timestamp(new GregorianC                 ;
                  m.setTime(timestamp.getTime());
               }
               catch ( Exception e )
               {
               }
            }
         }
         counter++;
      }
   }

// The loadMachines method is used to create all the machine                heir
// information from a textfile passed into the method, and t                ssed
// in with three hundred machines.   It then sets those machines to the values given in the
// textfile, using their number within the lab to choose their location in the vector.

public void loadMachines(String filename)
{
    try
```

File: Seeker.java
Method: loadMachines

Last Revision
Revision: 4
Revised: 13-08-2006 15:16
Revisor: rst498
Comment: The machine info is now loaded from the input files and loaded into the...

Previous Revisions
Revisions: 4
Frequent Revisor: rst498
First Added: 10-08-2006 11:32

Figure 2 - Example of how awareness entity could be represented using a pop-up information box

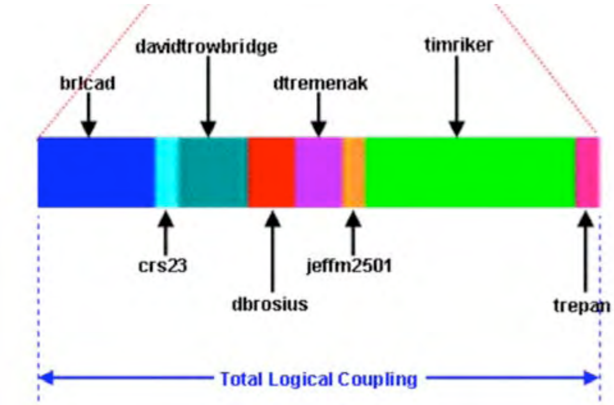


Figure 3 - Visualization of an awareness entity: developer activity contributing to coupling is represented by coloured bars

4. CONCLUSION

Using the architecture described for the development of tools supporting team awareness provides a number of advantages. Firstly, it provides a separation of concerns between the task of gathering awareness data, and presenting it to the user. The awareness entity allows for the data to be presented in a multitude of formats, without having to change the method in which the data is retrieved. This separation of concerns also allows support tools to be easily adapted to harvest awareness facts from different kinds of awareness resources.

Another advantage of using such a design in awareness tools is that the user will be presented with relevant information more often, and not overloaded with awareness information pertaining to unrelated artifacts. The awareness entities contain information related to a specific artifact, and so the user will not have to search through many unrelated items of information.

When designing tools to collect information concerning the activities and expertise of others, a number of privacy concerns are raised. For example, sometimes it is not desirable to reveal the intricate social networks that sometimes develop in large teams. It can become apparent that certain individuals are known for making faulty changes, or giving bad advice on the mailing lists. Consideration must be taken not to mine data that may be damaging to a project or a person's privacy.

In summary the technique presented in this paper should result in tools that require the developers to spend less time maintaining awareness through the checking and browsing of textual channels, yet still allow them to be aware of the information that is being posted to these resources. While it has been shown that open-source projects rely heavily on textual channels for maintaining awareness [3], it is conceivable that commercial projects may use similar means to keep aware of team activities. Further study may be needed to determine this, however our architecture could also be adapted to non-textual channels of awareness information.

5. REFERENCES

[1] Cubranic, D., Murphy, G.C., Singer, J., and Booth, K.S. Learning from Project History: A Case Study for Software Development. *Proc. CSCW*, 2004, 82-91.

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

[3] Gutwin, C., Penner, R., and Schneider, K. Group Awareness in Distributed Software Development. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, 2004, 72-81.

[4] Gutwin, C., Schneider, K., Paquette, D., and Penner, R. Supporting Group Awareness in Distributed Software Development. *Proceedings of IFIP Conference on Engineering for Human-Computer Interaction*, 2004.

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

[6] Kirsh-Pinheiro, M., Valdeni de Lima, J., Borges, M.R.S. A Framework for Awareness Support in Groupware Systems. *The 7th International Conference on Computer Supported Cooperative Work in Design*, 2002.

[7] De Souza, C., Froehlich, J., and Dourish, P. Seeking the Source: Software Source Code as a Social and Technical Artifact. *Proceedings of the ACM Conference Supporting Group Work GROUP 2005*.

[8] Viégas, F. B., Golder, S., Donath, J. Visualizing Email Content: Portraying Relationships from Conversational Histories. *CHI* 2006.

Together apart: An ethnographic study of industry-academia collaboration

N. Sadat Shami
Cornell University Information Science Program
301 College Ave, Ithaca NY 14850, USA
sadat@cornell.edu

ABSTRACT

Industry-academia collaboration brings together collaborative partners that have very different cultures, goals and technologies. Through an ethnographic study, this paper highlights various organizational constraints and their impact on collaboration effectiveness. Team characteristics including multiple group memberships and geographic distribution coupled with stringent security policies and lack of explicit collaborative norms can all cause impediments to effective collaboration. Implications for the design of collaboration technologies to cater to these organizational circumstances are discussed.

Author Keywords

Industry-academia collaboration, collaboration technology, collocation, ethnography, organizational culture.

ACM Classification Keywords

H.1.2 [**User/Machine Systems**] *human factors*; H.5.3 [**Group and organizational interfaces**] *computer supported cooperative work.*

INTRODUCTION

Collaboration between industry and academia creates great opportunities for sharing knowledge, accessing scarce expertise, making efficient use of limited resources, and leveraging funding opportunities from government agencies [9]. This is achieved through access to the skills and capabilities of each partner in the collaborative relationship. Recent advances in computer and communication technologies increase the potential for collaboration between industry and academia members that do not share geographic proximity.

Inspite of technological advances and the benefits of industry-academia collaboration, such partnerships face considerable challenges [9]. These challenges are compounded when participants are geographically distributed. It has long been recognized that the real complexities of distributed work lie in the interactions between people, i.e., how they communicate, how they share information, and how they negotiate shared goals. These human interaction aspects can turn even the simplest of tasks into very complex operations. Consequently CSCW researchers have studied social properties of distributed teams such as trust [3], conventions [4], common ground [5], attribution [1] as well as the technologies to support such teams (see [6] for a review). Although these studies involved distributed teams, we feel that industry-academia collaboration has certain characteristics that make it particularly interesting. Different goals, different cultures, and different technologies set industry academia collaboration apart from other collaborative practices where organizational differences are not as pronounced.

While the literature on collaboration is vast, there have been relatively few studies of industry-academia collaboration. The majority of those studies have focused on issues such as knowledge transfer from scientific or engineering labs in the university to industry [8]. In 1994, Slonim et al. introduce the Center for Advanced Studies (CAS) model of applied research involving industry and academia members [9]. In a later study, Perelgut et al. reflect on the initial model and evaluate its success [7]. These two studies detail principles and strategies that provide a solid basis for undertaking industry-academia collaboration. However, to the best of our knowledge, there have been relatively few published accounts of critical analysis of practice. Such informed ethnographies provide a rich and nuanced description of the subtleties inherent in industry-academia collaboration. They also provide developers of CSCW systems with an understanding of user behavior as it actually exists, rather than how it ‘ought’ to be. In this paper, we provide an ethnographic account of four distributed teams comprising of industry-academia members engaged in software development. As such, the material presented here documents the varied and complex interrelationships among individuals in work teams participating in industry-academia collaboration.

STUDY SETTING

This study was conducted at a software development lab of TC3 (a pseudonym), one of the largest information technology companies in the world. The lab is the third largest within TC3 and leads development efforts for a large number of software products.

The mission of the research wing of the lab is to bring together university faculty, graduate students, and TC3 developers to work on projects that are mutually beneficial to TC3 and the academics. These projects are 2-3 years in length. Students and faculty divide their time between the lab and their university over the duration of the project. Most students spend at least 3 months of the summer at the lab where they work closely with TC3 developers. During that time, their faculty advisor makes occasional visits. When their classes start, students return to their universities. They continue to work on the project, but more closely with their faculty advisor. A TC3 Research Liaison physically located at the lab is assigned to each project and performs project management duties, even when students and faculty are away at their universities. The Liaison is the connection between TC3 and the academic community and is responsible for ensuring the project meets its objectives.

We were provided with an opportunity to analyze work patterns and use of collaboration technologies of four (out of over 40) representative project teams. All teams were working on a particular version of a major software subsystem. The teams were at different stages of their respective development cycles and consequently no overarching process was applicable to all teams. Each team was responsible for mandating its own processes. Teams ranged in size from 6 to 9 members.

Data for this study was collected over four months of the summer. This is a period of time when most student members of project teams are physically located at the lab. Students have their desks in an open office environment within the research wing of the lab. This space is known as the student collaborative space. Desks are arranged in clusters of four and are partitioned with walls. The height of the walls allows some degree of privacy. Such a setting is meant to facilitate ease of communication and interaction. Research Liaisons have their cubicles adjacent to the student collaborative space. The cubicles of TC3 developers are located on different floors of the lab.

METHODOLOGY

The data collection procedures used in this study were participant observation, semi-structured interviews, and email analysis. Data collection was performed during my stay at the lab. I was physically located in the student collaborative space. I had a Liaison that supervised me and I went through the same activities as students from other project teams. This allowed me to develop an in-depth understanding of the work context by casting me as both the informant 'insider' and the analyst 'outsider'. Being an authentic participant in the study allowed me to draw on my personal experiences, thoughts, and reactions in addition to the collected data. The hallway conversations I had with other students, the informal interactions during picnics and socials, the research talks I attended with other students and TC3 employees all allowed me to develop a rich appreciation of the teams I studied.

While at the lab, I spent roughly forty hours of observing meetings of the four project teams. Additionally, being collocated with the students of the project teams I was studying allowed me to observe their natural work settings. This provided useful contextual information about who were dominant members of the teams, how they addressed bottlenecks, and how they interacted with each other.

I interviewed *all* members of the project teams I was studying. In total, thirty one semi-structured interviews were conducted to probe individual impressions about collaborative work processes. Each interview typically lasted an hour. Interview questions revolved around project responsibilities and goals, relationship with other project members, progress of the project, physical arrangement of project members, technologies used and strengths/weaknesses of the project.

Participants were requested to copy me on non-personal project related email. This provided insight into the interaction and communication patterns among team members. However, obtaining these emails was not an easy task. Project participants were at times forgetful in copying me on their emails. I would then meet with those participants and have them go through their 'sent messages' folder and forward relevant emails to me. I would do this on roughly a bi-weekly basis when I noticed a lack of incoming email from project members or when I received an email that had content that did not follow previous emails.

Being physically located at the lab was not without its challenges. I sometimes faced a difficult environment when I started conducting research. My lack of experience and perhaps low credibility with employees all contributed to difficulties in conducting valid and probing research. This field study was particularly tricky since project participants were told that the researchers were interested in identifying team processes that were working, as well as *not working*. Collection of such data may be perceived to be intrusive or sensitive. While management desires results that translate into answers, because of privacy laws they are unable to grant outsiders access to some of the very data that could result in effective solutions. On the researchers' side, there is a risk of upsetting key stakeholders by revealing unfavorable information that may result in a curtailment of future access to the study setting. In order to conduct such potentially sensitive research, I had to gain the trust of study participants so they would confide in me. Management also had a true desire to implement changes in order to improve collaboration effectiveness and was thus convinced that they needed to know what was going wrong, regardless of whether it was pleasant to hear.

Being physically collocated at the lab was particularly useful in establishing trust with research participants. Being able to have face-to-face contact with other students, Liaisons, and TC3 developers allowed me to establish rapport.

The data gathered from participant observation, interviews and emails overlapped, thus providing a way of triangulating, or cross-checking the accuracy of the data. Grounded theory [10] was used to identify themes (patterns of recurring phenomena) in our observation, interview and email data. Through open coding we first identified the general categories of: project deliverables, impediments to deliverables, technologies used, organizational policies and social norms. Axial coding was then used to search for causal relationships between these concepts. Through selective coding, we refined these concepts into an understanding of how organizational constraints affected collaboration: how multiple team memberships created different priorities, how project teams stagnated because of differing priorities, how security policies could limit the natural flow of collaboration, and how the lack of norms affected sharing knowledge.

RESULTS

Our analysis revealed a complex relationship between team characteristics interacting with social norms, and technology use. We identified both positive and deficient group processes. However, we chose to focus on deficient processes in order to discuss how collaboration technologies may ameliorate them. The research model employed by the lab has been in place for several years and has resulted in the creation of a community of faculty performing research that is of mutual interest to TC3 and themselves. This is a testament to the success of the research model.

Multiple memberships and differing priorities

All four categories of members in a project team have multiple group memberships. TC3 developers are involved in different product teams, Liaisons have an average of 8 project teams they oversee, faculty members have other research and students they have to supervise, and students have their coursework and dissertation to take care of. Most challenging for the project team is multiple memberships of TC3 developers. Their availability is often dictated by product release cycles, which can take priority over the research project. In two of the projects we studied, there was considerable inertia because all TC3 developers were busy with their primary responsibility of meeting production deadlines. Furthermore, since research projects are based on problems that are not on the critical path of development products, there is some degree of uncertainty regarding whether the research project will be able to be incorporated into a TC3 product. In some cases, the absence of development leadership slowed projects, and in some cases they even stagnated. The following quote illustrates the tension of multiple memberships and priorities.

"Anyone who is not personally invested in the research project will continue to struggle with the balance of giving it enough focus… we have a lot of tugs, pressures and pulls and this is one of the first things to fall off the cart when things get heated up."

Because of these pressures, it is hard to keep track of the current status of a particular project, especially for TC3 developers and faculty members.

Security policies and geographic distribution

Security policies required that most project members be on site when accessing TC3 services. Once outside the premises of the lab, students and faculty members did not have access to TC3 email or instant messaging applications, let alone the code they were working on. The imposed restrictions on use of collaboration technologies impeded the natural free flowing nature of collaborative work. Students could not work from home. When they returned to their universities, they could not easily access their work and maintain contact with TC3 developers. The instant messaging application was the most popular means of communication among both students and TC3 developers. It provided awareness of team members and afforded opportunistic interactions. Needless to say, the use of this technology was sorely missed by students when they were not physically located at the lab. On the other hand, TC3 employees had Virtual Private Network (VPN) access so they could access all lab resources when they worked remotely. Out of approximately forty students, only one was given VPN access privileges.

Even when on-site, mandatory security policies could create impediments to work processes. One of the project teams had a faculty member located in Japan. Coincidentally, a student member of the team was working on an audio conferencing tool and the team decided to use it for conferences calls with Japan. However, when it came time for the meeting, team members could not connect with Japan after repeated tries. It was later discovered that the reason behind this was that the TC3 firewall did not allow connections to the conferencing server, which was located at the student's university.

Norms of effective collaborative work

Perhaps the most interesting finding of our study was that adequate norms of effective collaborative work could have led to more opportunities for sharing knowledge. The collaborative student space where students sit is an ideal setting for creating a community of practice that can serve as an additional source of support and learning. The intention of having students sit together in close proximity was so that they could informally consult each other and get exposed to new and diverse ideas. Cultivating explicit norms emphasizing the collaborative nature of the space would help students take better advantage of collocation. *"Most students just do their own thing"* was a comment by a student. For many students, the collaborative space was unlike anything they experienced before. Fostering norms encouraging exchange of ideas would thus promote better use of the collaborative space for knowledge sharing.

Nurturing collaborative norms might also lead to a less competitive culture, as was evident in one project team. That team had students from two different universities. In contrast to working together collaboratively on the same project, one student commented:

"It seems that there are two teams, and goals differ between the two. Ours is to complete our tasks, while theirs is to complete theirs. Two separate teams with minimal sharing of information."

IMPLICATIONS FOR THE DESIGN OF COLLABORATION TECHNOLOGIES

The immediate pressures of daily production tasks and deadlines will dominate employee decisions on how they allocate their time. Thus collaboration technologies need to allow individuals that have multiple group memberships a mechanism to stay abreast of the projects they are involved in without increasing cognitive burden. Since scheduling meetings is often difficult, a shared workspace can act as a *supplement* to meetings. Project members can input what they did during the week, where they are facing bottlenecks and what they need from fellow members. This asynchronous medium would allow project members to check on project status and remain 'in the loop' while allowing them to concentrate on their primary responsibilities.

The criticism with shared workspaces is that people will not take the time to check it. However, recent studies show that subtle design changes can influence behavior, such as making employees aware that their supervisor is checking the system [6]. A study of an online knowledge sharing community for teachers found that it was successful even in the absence of a strong organizational mandate [2]. Volunteer knowledge stewards created conditions necessary for the success of the community by playing an active role. In an industry-academia project team, students can act as knowledge stewards to keep the shared workspace running smoothly.

Collaborative technologies can also help in creating collaborative norms. Personal profiles in a digital space where students share their background and interests might facilitate interaction between students by providing a common ground for conversation. Students can browse profiles of other students before they even arrive at the lab and find others that share their interests. This provides a springboard for conversation that may lead to exploring common or related interests.

There is also a need to create secure collaboration technologies that can be used anytime, anywhere. Such technologies will allow geographically dispersed teams to adhere to security concerns while preserving the open and flexible nature of collaborative work.

CONCLUSION

Our findings lead us to believe that while the model of industry-academia collaboration studied was very effective overall, overcoming organizational constraints could improve it even more. Better alignment of priorities, less stringent security policies, and fostering collaborative norms can help better realize the benefits of collaborative relationships. Our study provides practical implications how collaboration technologies can be better designed to cater to different organizational circumstances.

ACKNOWLEDGMENTS

Nathan Bos supervised this work. It was funded in part by TC3 as well as NSF grant #IIS 0308009 to Judy Olson and Nathan Bos.

REFERENCES

1. Cramton, C. Attribution in distributed work groups. In P. Hinds & S. Kiesler (Eds), *Distributed work: New ways of working across distance using technology*, MIT Press, 2002.
2. Brazelton, J. & Gorry, G.A. Creating a knowledge-sharing community: If you build it, will they come? *Comm. of the ACM,* 46(2), (2003), 23-25.
3. Jarvenpaa, S., Leidner, D. Communication and trust in global virtual teams. *Journal of Comp. Mediated Comm.*, 3(4), (1998).
4. Mark, G. Conventions and commitments in distributed groups. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 11 (3-4), (2002).
5. Olson, G.M. and Olson, J.S. Distance Matters. *Human Computer Interaction*, 15, (2000), 139-178.
6. Olson, G.M. and Olson, J.S. Groupware and Computer-Supported Cooperative Work. In J. Jacko and A. Sears (Eds.), *The Hum. Comp. Inter. Handbook.* LEA, 2003.
7. Perelgut, S.G., Siberman, G. M., Lyons, K. A., Bennet, K. L. Overview: The Centre for Advanced Studies. *IBM Systems Journal*, 36 (4), (1997).
8. Scott, J.E. & Gable, G. Goal Congruence, Trust, and Organizational Culture: Strengthening Knowledge Links. In *Proc. ICIS '97*, (1997), 107-120.
9. Slonim, J., Bauer, M.A., Larson, P.A., Schwarz, J., Butler, C., Buss, E.B., Sabbah, D. The Centre for Advanced Studies: A model for applied research and development. *IBM Systems Journal*, 33 (3), (1994).
10. Strauss, A.L. & Corbin, J. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications: 1998.

Help, I Need Somebody!

Computer-mediated Preemptive Mentoring for Domain Novices

Andrew Begel
Microsoft Research
One Microsoft Way
Redmond, WA 98052
andrew.begel@microsoft.com

ABSTRACT

Information discovery is a very difficult and frustrating aspect of software development. Novice developers are often assigned a mentor who preemptively provides answers and advice without requiring the novice to explicitly ask for help. A similar situation occurs among expert developers in radically collocated settings. The close proximity enhances communication between all members of a group, providing needed information, often preemptively due to ambient awareness of other developers. In this paper, we propose a mechanism to extend this desirable property of preemptive mentoring to developers in more traditional software engineering environments. The proposed system will infer when and how a developer becomes blocked looking for information, and notify an appropriate expert to come to his aid. We believe that this preemptive help will lower developer frustration and enhance diffusion of expert knowledge throughout an organization.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications

Keywords

expert, novice, mentor, blocking, information discovery

1. INTRODUCTION

Information discovery is one of the most difficult, frustrating tasks in software development. Experienced developers often become blocked on fairly easy-to-formulate, but difficult-to-answer questions concerning code rationale, bug triage, and co-worker awareness. Novice developers, whether they are fresh out of university or transferred in from another professional job, also experience the same kinds of frustrations in information-seeking, but have a unique aid, a mentor. Mentors are experienced developers, or domain experts, whose job is to look over their protégés' shoulders and help them out when they become confused or blocked, sometimes before they have even asked for help. Once the novices gain experience, the mentoring relationship is tailed off and they are left to her own devices.

Berlin and Sim and Holt [1, 16] characterize the mentor relationship provided to new hires or "immigrants" to a software team as providing answers to simple questions, explaining design rationale and hard-to-find information, proffering advice on tool usage and administration, and very importantly, introducing them to their new social network. They and Singer and Lethbridge [17] find that the mentor relationship tails off after a few months once the novice becomes integrated into the group. This is a good thing because in many development cultures, it can be perceived as intrusive to continually ask questions of others, though providing answers and advice can usually be considered to be part of one's job and helpful to advance the product as a whole [12].

Once a novice learns how to find his way in an organization, is there no more help to be provided? Of course not. Numerous studies of information-seeking behaviors show that coordination between software developers goes on at various levels of an organization [1, 2], through various communications modalities [6, 13, 20], enabling developers to discover several important classes of information [8, 9] and promote team awareness [5, 6, 9, 11, 14, 18]. These studies on information-seeking behaviors were characterizing experienced developers, not novices. Who are these human sources of information and what is their relationship to one another? Do domain experts have mentors?

Part of the problem with associating a mentor with a software developer is that experienced developers create and maintain code with a large and mostly unique domain due to the traditional approach to divide up software projects by contributor. While a mentor for a novice need only be an expert in the local group's software project (since novice projects are chosen to be small and self-contained), a mentor for an expert might need to be quite knowledgeable about an entire product's codebase. This may be possible in a small organization, but it cannot scale to large ones with many software development teams. Thus, an expert is likely to require multiple experts, each with expertise in a particular area.

So, to where do real experts turn for help when they get stuck? Ko, DeLine and Venolia conducted a study of the content of information sought by 17 developers at Microsoft [8], and noting where the answers (if any) were found. Their data indicate that many types of information that cause developers to become blocked could be found by going to coworkers. The study did not identify the responders' areas of expertise, nor place them in the social network of the seeker. In open source projects, experts turn to mailing lists to look for other experts [5]. An inquiry is made to a mailing list, and the experts monitoring the list will see the question and respond when they have the answer. Other researchers analyze software repositories to automatically identify likely experts given a domain, for instance, a code file or module in a project [10]. Those who edit a file or module most often are inferred to be most expert in that area. Since this information is fairly well hidden without analysis, de Souza et al. [3] propose a social call graph (analogous to a procedure call graph in program analysis) that relates developers to one another when their code interacts in some way.

These notions of code expertise are all based on the notion that an inquirer will seek out an expert when he gets blocked or stuck. But that is not always what happens. Latoza, Venolia, and DeLine report that developers first exhausted other, often inadequate, sources of information (the code, documentation, debuggers, logs, bug databases), before seeking the help of others [9]. Sillito, Murphy and De Volder note that the questions developers often ask do not always map very well onto the answers that software tools can provide [15]. Humans can be much more efficient at answering vague or desperate questions. Mentors assigned to novices keep track of them and drop by their desks to see what they are up to. Mentors can usually tell when a novice is stuck without the novice having to ask. The mentor *preemptively* provides the answer before the novice wastes too much time looking on his own. This is a good thing. Unfortunately, in a typical development culture of private offices and cubicles, looking over someone's shoulder to see if they are stuck is not prevalent.

Teasley et al. [19] report that a technique called radical collocation makes preemptive advice a regular occurrence. Radically collocated groups work together in one big room. Developers can use their ambient senses to overhear conversations and see their colleague's screens as they work. Whenever one developer needs help, she needs only pop up her head and spot the right person already in the room to answer her question, or someone else will notice her frustration and preemptively ask to help out. If another expert is nearby, he can join the conversation just as easily and provide extra information, context and institutional memory. Radical collocation, in short, provides the mentoring relationship that novice developers receive and makes it available to every developer in the room.

Unfortunately, just like most face-to-face communication, radical collocation induces large coordination problems when scaling to very large software projects. In the past, when direct human coordination proved unwieldy, researchers developed technological solutions to mediate the communication, such as email, bug databases, configuration management systems and wikis. Using each of these systems may appear to each developer to be a locally optimal solution, however, they exact a cost. Each provides a less immediate and lower bandwidth mode of information transfer than would otherwise be achieved with face-to-face communication.

We think that face-to-face communication opportunities should be encouraged, mediated by a new technology that combines the best aspects of radical collocation, social call graphs and ambient displays. This technology would enable experts across a large product team to preemptively interrupt domain novices when they are stuck on a problem, without requiring the novice to discover who to ask, without requiring the novice to exhaust all personal means of searching information repositories before asking his question, and without a novice feeling like he creates too much of an imposition on the expert to ask his question.

In our model, the expert is considered the altruistic, omniscient superhero who comes to save the day when he somehow detects that another developer is in trouble. It is a scenario already proven to work for novice developers new to a programming team, and in radically collocated teams for experts who need help. It is a model that can coexist with notions of privacy where individual developers maintain their own office or cubicle space. We view our particular statement of the model in direct opposition to an alternate one, where a system notices that a developer is stuck on something and "warns" the expert that the novice may come to ask a question. By using the word "warn" we mean to imply that an introverted expert may actually close his door to maintain privacy or appear very busy to avoid taking the novice's question. It is exceedingly important, we think, to ensure the model is one of altruism, giving advice to a customer in need, rather than bothersome interruption, receiving questions from someone who does not deserve answers or who should have been smart enough to figure out the answer.

Why should an expert be so altruistic and preemptively talk to someone who needs their help? Experts are short of time; they need to get their own work done [12]. But, domain novices who use an expert's code are the expert's *customers*. If the customers cannot use the expert's software, they will feel frustrated and spread their negativity to their friends. If the customers get blocked and the author of some code comes to their rescue, a positive review can be formed and spread. In addition, creating satisfied customers reflects well on a developer among the members of his own product group, as long as they all know about it.

Note that not all questions require human help. If a system can identify a human developer as an expert in a particular area, it ought to be possible to tag other information sources as appropriate repositories of answers that a domain novice might look at before needing help. In fact, frequent use of these sources could be interpreted as a trigger to understand when the novice requires expert intervention.

To test out our ideas, we will need to answer five questions.

1. Is it possible to tell when a developer is stuck or blocked and needs help? It is likely a domain expert can tell, but can this state be inferred through logs of developer actions?

2. Once it is possible to know when someone is stuck, is it possible to identify what topic or code area the developer is stuck on? It may be possible to record wear on the code, documentation or bug database to detect this.

3. Once the areas of blockage are known, is it possible to use it to discover which the likely experts who know something about the areas? Mining source code repositories for experts based on code ownership [10] is a start, but should be extended to other information sources as well as validated in a real software project.

4. What kinds of ambient displays can you put on a developer's desktop to make them aware when people need their help? For example, a digest of all domain novice/customer activity could be delivered to the expert (and his product group), enabling the expert to understand his customers' behaviors, spot when they are stuck, and preemptively help them when it appears appropriate.

5. How would such a technology affect the culture of the organization in which it was deployed? The design and potential success of such a technology has to be sensitive to an organization's existing culture, especially in regards to the value system placed on asking questions, asking for help, helping someone in need, and helping someone repeatedly. A thorough understanding of these issues can be developed using a value-sensitive design methodology [4].

2. STUDY PROPOSAL

To learn if our model is viable and answer each question posed above, we will undertake several studies. The first study will begin with an survey of a random sampling of software developers at Microsoft. Surveyed developers who agree to take part in the study will be shadowed for an entire workday once a month by a researcher who will code their activities according to a coding schema initially designed by Ko, DeLine and Venolia [8]. This schema will enable us to record developers' information-seeking activities and their outcomes. If other developers meet the study participant to ask a question, those interactions, the identity, and the expertise of the coworker will be recorded as well. To capture more information about developer behavior, a developer's computer-based activities will be logged. An IDE logger will record their development activities, a window title logger will record their windowing behavior [7], and if accepted by the developer and his colleagues, an email logger will record his conversations with other members of their software team. Our hope is that the logging information can be used to synthesize a summary of developer behavior that can be subsequently analyzed and correlated with the observations to enable us to infer the tasks the developer is working on and identify the events that lead up to task switches caused by blocking.

The second question can be answered by the shadow observer during the first study. Whenever a task switch due to blocking occurs, the observer can ask what areas of the code the developer was working in and learn whether they are related to the blockage or are merely incidental. This can then be correlated with the logging information.

The third question can be answered with a survey. Given several existing technologies for relating developer experts to code and bug reports, a list of potential matches can be generated. We can send out a survey to developers at Microsoft and ask them what their area of expertise is and for which files, code modules, and bugs they feel they could provide expert help. There is quite likely to be a many-to-one mapping of expert developers to area. A question we might ask next is how often the mapping remains stable or changes over time. It is possible that human resources information might be used to help keep this mapping up to date.

Finally, the real test is to build a system using the task inference technologies proposed above that can notify an expert automatically as to the activities of the domain novice who is using his software, and enable the expert to stop by or communicate electronically to solve the problem. Note that this kind of interaction can be run through a Wizard of Oz study, where an expert observer can watch a developer during the day and hit a private **Help Me** button when they think the developer has become stuck. We expect to provide a knob to the developer to tune how quickly help is requested after they get stuck. Some developers may want more time to play around before someone helps them, some may want less.

The effects of this technology may be difficult to measure. Much of it may simply be that the general stress level and level of frustration experienced by developers goes down with this tool. It may take just as long to get help as it did before, but as developers become more accustomed to asking for (and providing) information to others, less time may be wasted learning about irrelevant code in the search for understanding. In addition, by linking domain experts with other programmers in the organization, knowledge will flow more freely.

The technology may have negative effects as well, including the perception that person who needs help is less capable than others, or that a person who offers help is not spending enough time doing their own work. A study is necessary to understand and evaluate the software development culture and identify how to design and sell the technology to make sure it is perceived as a good thing and not a target or enabler for scorn.

3. CONCLUSION

In this paper, we have proposed a new mechanism for enabling expert developers to preemptively help less knowledgeable colleagues when those colleagues get blocked. Carrying out the proposed studies will help inform us of the feasibility of this mechanism and of its utility and acceptance in practice. If it works, it could help bring some of the benefits of group awareness and participation enjoyed by small development teams to larger organizations.

4. REFERENCES

[1] L. M. Berlin. Beyond program understanding: A look at programming expertise in industry. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer, editors, *Empirical Studies of Programmers: Fifth Workshop*, pages 6–25. Ablex Publishing Corporation, 1993.

[2] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, Nov. 1988.

[3] C. R. B. de Souza, D. F. Redmiles, L.-T. Cheng, D. R. Millen, and J. F. Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 63–71. ACM, 2004.

[4] B. Friedman. Value-sensitive design. *interactions*, 3(6):16–23, 1996.

[5] C. Gutwin, R. Penner, and K. A. Schneider. Group awareness in distributed software development. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 72–81. ACM, 2004.

[6] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 21–24. ACM, 2004.

[7] D. R. Hutchings, G. Smith, B. Meyers, M. Czerwinski, and G. G. Robertson. Display space usage and window management operation comparisons between single monitor and multiple monitor users. In M. F. Costabile, editor, *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28, 2004*, pages 32–39. ACM Press, 2004.

[8] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development groups. In *Submitted to 29th International Conference on Software Engineering (ICSE 2007).* ACM, 2007.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 492–501. ACM, 2006.

[10] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 503–512, New York, May 19–25 2002. ACM Press.

[11] C. O'Reilly, P. J. Morrow, and D. W. Bustard. Improving conflict detection in optimistic concurrency control models. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, May 14-15, 2001 and Portland, OR, USA, May 9-10, 2003. Selected Papers*, volume 2649 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2003.

[12] L. A. Perlow. The time famine: Toward a sociology of work time. *Administrative Science Quarterly*, 44(1):5781, 1999.

[13] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.

[14] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *ICSE*, pages 444–454. IEEE Computer Society, 2003.

[15] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006. ACM SIGSOFT.

[16] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *ICSE*, pages 361–370, 1998.

[17] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *IWPC*, page 173. IEEE Computer Society, 1998.

[18] M.-A. D. Storey, D. Cubranic, and D. M. Germán. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In T. L. Naps and W. D. Pauw, editors, *Proceedings of the ACM 2005 Symposium on Software Visualization, St. Louis, Missouri, USA, May 14-15, 2005*, pages 193–202. ACM, 2005.

[19] S. Teasley, L. Covi, M. S. Krishnan, and J. S. Olson. How does radical collocation help a team succeed? In *CSCW*, pages 339–346, 2000.

[20] J. Wu, T. C. N. Graham, and P. W. Smith. A study of collaboration in software design. In *ISESE*, pages 304–315. IEEE Computer Society, 2003.

Exploring the Relationship between Software Dependencies and the Coordination of Software Development Work

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

Abstract

In this paper we present a tool to facilitate the work of managers of global software development projects. This tool explores the relationship between software dependencies and coordination of work and uses social networks to suggest potential coordination problems for managers. The overall architecture of the tool is described as well as the theoretical and empirical motivations for the tool.

***Keywords**: Software Dependencies, Coordination, Distributed Software Engineering, Social Networks.*

1. Introduction

In the past years, more and more organizations have distributed their software development projects in sites around the globe. Different strategies, tools, and approaches have been proposed to facilitate this scenario which faces social, technical and cultural challenges [1]. In this paper we present a tool that aims to facilitate the work of managers of global software development projects. This tool is based on theoretical predictions and empirical observations about the nature of software development work. Through our tool, a manager could potentially monitor distributed software developers and anticipate coordination problems among them. This paper describes the design of this tool and our theoretical motivations.

2. Background and Motivation

One way to manage the growing complexity of complex systems is to decompose them into smaller parts, the subsystems [2]. In software systems, this idea is called decomposition, and these smaller parts are called modules [3]. The predictable consequence of dividing a system into modules is that these modules need to be put back together in a way that the software system can provide its services. A dependency between software modules is said to exist when a module relies on another to perform its operations or when changes to the latter must be reflected on the former [4].

Software engineers have long recognized the need to deal with dependencies. For example, there are different techniques for program dependency analysis [5]. These approaches are used, among other goals, to improve software testing, evolution, and understanding.

Another approach is the creation of mechanisms in programming languages to reduce dependencies between software elements. In this case, the most important principle is Parnas' information hiding [6]. Parnas proposed more than just a technical approach: he recognized the relationship between software dependencies and coordination when he suggested that by reducing dependencies between modules, it is possible to reduce developers' dependencies on one another, a managerial advantage. Nowadays, this is a well-known argument cited in software engineering textbooks [3, p. 241]. Conversely, but also supporting this relationship between dependencies and coordination, Conway [7] postulated that the structure of a software system would reflect the communication needs of software developers. In short, whereas Parnas argues that dependencies *shape* the coordination and communication activities performed by software developers, Conway argues the converse: that dependencies *reflect* these coordination and communication activities. That is, technical dependencies between components create a need for communication between developers, and similarly, dependencies between the development tasks are reflected in the software. Both Parnas' and Conway's arguments have been validated by different empirical studies in collocated and distributed software projects [8, 9].

This relationship between software dependencies and the coordination of the work holds even when modular decomposition is applied. In fact, software engineering research has already found that one module can not be implemented completely independent of its clients [14]. This means that software developers who are supposed to work independently instead need to communicate and coordinate to guarantee a smooth flow of work. Our own empirical studies acknowledge this when they describe the coordination problems faced by software developers despite the usage of interfaces and state-of-the-art software development and collaborative tools [9].

Despite this acknowledged relationship between dependencies and communication and coordination needs, this relationship has not been explored to facilitate and understand software development activities. Software development is a strong candidate for exploring this relationship since (i) dependencies among software components can be automatically identified, and (ii) software is malleable, i.e., their dependencies, if so desired, can be more or less easily changed, and consequently the coordination of those developing it. In this paper we explore this relationship to support software development projects through the description of a software development tool to help managers of distributed projects. This tool is described in the following section.

3. The Tool

Among the empirical studies that describe the relationship between software dependencies and coordination, some of them are more relevant to this work. To be more specific, Morelli [10] and Sosa [11] found a strong correlation between dependent components in a software system and the frequency of communication among the members dealing with these components. This suggests that developers dealing with dependent components are more likely to engage in communication than developers implementing independent components. According to these authors, technical dependencies could then be used to predict communication frequency among team members. That is, given two dependent software modules, the developers responsible for developing those modules need to interact to coordinate their work, despite the usage of interfaces and other mechanisms to minimize dependencies. It is this insight that guides the design and usage of our tool.

However, in order to explore the relationship between software dependencies and coordination, it is necessary to identify dependent pieces of software and communication events among the software developers. Our tool automatically identifies dependent pieces of code and their authors using Ariadne [12], so that it is possible to create a social network [13] of developers that establishes which developer depends on the code of another software developer. Communication events are more difficult to be identified since developers can use different media to communicate: emails, instant messages, phone calls and so on. Our current implementation registers email exchanges through an event-notification server that receives all emails exchanged among pairs of developers. The aggregation of all emails creates a communication network of developers. These two social networks – dependency and communication – are combined by our tool.

The goal of our tool is to automatically identify situations when there is a *mismatch* between the dependency and communication networks. This includes two situations. In the first case, there is a dependency between two components, but the software developers dealing with them are not engaging in communication events. This might mean that those developers are not aware of each other, a usually problematic situation [9]. The second case happens when two developers are communicating with some frequency but there is not a dependency between their components. This situation might suggest a need for re-structuring the architecture of the system (that's why they are communicating) or that possibilities for software reuse are being lost.

Our tool supports the visualization of different social networks: the communication and dependency networks, the network that highlights the matches between the communication and dependency networks, and finally, the network of communication (dependency) not accompanied by dependency (communication). This way it provides to managers easy access to the information of interest.

Figure 1 below presents three views provided by the tool: (i) the union of the two networks, (ii) the dependency network minus the communication network, and (iii) the communication network minus the dependency network. Again, the overall idea is to identify the mismatches between the networks, which is achieved by presenting the difference between the networks.

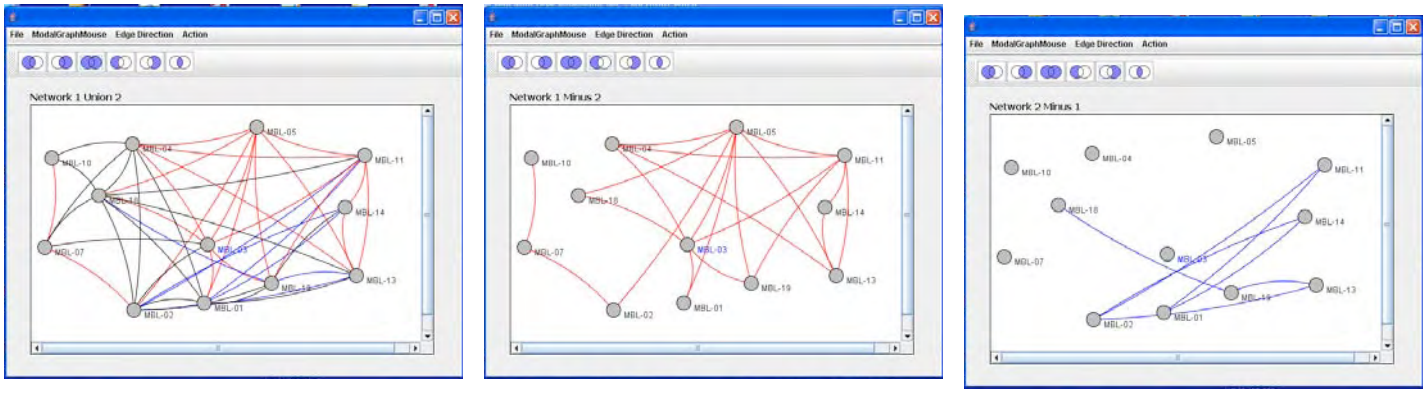


Figure 1: Tree different views provided by the tool

An important design decision of our tool is just to present information to the manager, letting him to decide how to handle it. That is, the tool just indicates the mismatches between the networks; it does not automate tasks for the manager. The reason is that the manager might be aware of additional information, which would help him to make sense of the reported mismatches: e.g., a refactoring of the code that is about to happen might justify the communication that is going on among developers despite the fact that their components are independent.

Our tool is implemented in Java and uses Elvin as the event-notification server that monitors and receives email information exchanged among software developers. The dependency network from Ariadne is imported as CSV files. The comparison between the social networks is done using matrix operations, as in standard social network software implementations [13] and the visualization of networks is done using JUNG (http://jung.sourceforge.net).

4. Conclusion and Final Remarks

The goal of this paper is to present a tool developed by the authors to facilitate the work of managers dealing with software development projects. This tool is based on the relationship between software dependencies and the coordination of software development work. This relationship has been predicted in the past and corroborated by different empirical studies. Managers could use the tool to monitor interactions among distributed software developers and therefore anticipate potential problems.

We plan to continue improving the tool and also to conduct empirical studies with it. For instance, by using it to analyze real data from global software development projects or deploying it in global teams.

5. References

[1] J. D. Herbsleb and D. Moitra, "Global software development," *IEEE Software*, vol. V18, pp. 16-20, 2001.

[2] H. A. Simon, "The Architecture of Complexity: Hierarchical Systems," in *The Sciences of the Artificial*. Cambridge, MA: The MIT Press, 1996, pp. 183-216.

[3] C. Ghezzi, et. al., *Fundamentals of Software Engineering*, Second Edition ed: Prentice Hall, 2003.

[4] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," in *Handbook of Software Engineering and Knowledge Engineering*, S. K. Chang, Ed.: World Scientific Publishing Co., 2004.

[5] A. Podgurski and L. A. Clarke, "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," Symposium on Software Testing, Analysis, and Verification, 1989.

[6] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *CACM*, vol. 15, pp. 1053-1058, 1972.

[7] M. E. Conway, "How Do Committees invent?" *Datamation*, vol. 14, pp. 28-31, 1968.

[8] A. MacCormack, et. al., Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, Harvard University, 05-016, 2004.

[9] C. R. B. de Souza, et. al., "How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development," presented at Foundations of Software Engineering, Newport Beach, CA, USA, 2004.

[10] M. D. Morelli, et. al., "Predicting Technical Communication in Product Development Organizations," *IEEE Trans. on Engineering Management*, vol. 42, pp. 215-222, 1995.

[11] M. E. Sosa, et. al., "Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry," *IEEE Trans. on Engineering Management*, vol. 49, pp. 45-58, 2002.

[12] E. Trainer, et. al., "Bridging the Gap between Technical and Social Dependencies with Ariadne," presented at Eclipse Technology Exchange, San Diego, CA, 2005.

[13] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge, UK: Cambridge University Press, 1994.

[14] Kiczales, G. Beyond the Black Box: Open Implementation. IEEE Software, 13 (1). 8-11.

Knowledge-sharing in open-source software development using a social bookmarking system

Davor Čubranić
cubranic@acm.org

1. INTRODUCTION

As increasing numbers of successful open-source software (OSS) projects reach hundreds of thousands of lines of code, a dominant architectural model has emerged that allows such projects to cope with communication and coordination challenges of working in geographically dispersed teams that are potentially open to anyone on the internet.

Despite their size, projects such as Linux,[1] Apache,[2] Mozilla,[3] and Eclipse[4] are driven by a small core group of no more than a couple of dozen developers. Although there may be numerous other contributors to the project, from bug reporters and testers to coders, the members of the core group typically write the majority of the system's code themselves and review all the externally written contributions before they can be accepted into the projects code base.

Thanks to the small core group size, these projects can maintain cohesion and focus even though their members may have barely even met face-to-face. However, the core group is also a limiting factor to the project's growth in a sense, since there is only so much its members can physically do in a given period of time. As a compromise between an ever-expanding list of feature requests from their users and the need to allocate the core group's time and effort, most large OSS projects have independently evolved an extremely modularized and extensible architecture. Such an architecture allows the project to keep the functionality written by the core team rigorously focused, while at the same time the wider developer community can develop third-party modules that plug into the core system and address their own specific needs.

Examples of such third-party modules range range from low-level device drivers in Linux, to Eclipse plug-ins that allow editing of more esoteric programming languages, to Mozilla extensions that change the web browser's user interface or add tighter integration with certain web services. The extensible architecture of these systems is arguably an important contributing factor to their popularity and growth of a lively user community. Indeed, it is hard to imagine that a successful open-source project of a reasonable size and duration and with a large user community could maintain centralized development. Analysis of communication patterns in over 150 OSS projects conducted by Crowston and Howison [1] supports this idea, finding that the level of centralization of communication in bug solving is negatively correlated with project size.

However, non-core developers interested in writing their own extensions face significant challenges. First, they have to understand the extension API and know the functionality that is available to extensions. Second, they have to know how to use the API properly, such as a particular sequence of steps that have to be executed in the right order. Third, developing an extension involves more than just writing the code and typically includes setting up its configuration files, packaging, and distribution. Each of these steps requires knowledge of specialized syntax and format, often idiosyncratic to the particular project. Therefore, a beginning extension developer must absorb a significant amount of very specialized knowledge that can not be found outside the project—for example, transferred from another project or even learned in school, the way a programming language might be. In other words, such a developer is completely dependent on the project's community while coming up to speed.

The community itself, of course, is initially completely dependent on the core development team to provide documentation and advice about the project. The importance of this initial step cannot be overstated: witness the problems faced by the Mozilla project in its first year, after Netscape open-sourced its browser code but was too slow to provide documentation that would allow beginners to the project to start contributing. [4] However, with time non-core developers start creating their own project documentation. Such documentation is usually written to complement the one provided by the core team, especially by being "friendlier" to newcomers to the project. For example, it may be written as a tutorial designed to get the newcomer off the ground writing useful code quickly, rather than as an exhaustive reference.

Open-source projects today use a variety of forums to which members can turn for help and information about the project, as well as discuss development issues or report bug. These forums range from official project web site, bug tracking database, and mailing lists, to IRC channels, blogs, and user-written newsletters. In this paper, we report on initial results of our investigation into a previously overlooked channel for knowledge sharing—web-based collaborative bookmarking.

2. SOCIAL BOOKMARKING SYSTEMS

Web browsers have early on introduced the "bookmarking" feature: creating a collection of pointers to web pages, together with addi-

[1] **–TODO: URL–**

[2] `http://www.apache.org`

[3] `http://www.mozilla.org`

[4] `http://www.eclipse.org`

tional user-defined metadata such as a brief description or a set of keywords. Social bookmarking systems [3] take this concept from a personal collection to one shared world-wide. Typically, a social bookmarking system allows individuals to enter their bookmark collection into the system, where it can be accessed by anyone on the internet, most commonly through a web-based interface. The power of social bookmarking comes from the multiple ways that their users can view, search, and organize the collection: by the bookmark's URL, author, or keywords. The system's home page is frequently the place where most popular bookmarks are listed, so that casual browsers come across them and further enhance their popularity.

The most popular social bookmarking system today is del.icio.us (pronounced Delicious).[5] One of the foremost features of Delicious is *collaborative tagging* – marking bookmarks with multiple keywords, or tags. The vocabulary of tags is completely open; any user is free to attach any word or combination of words to a bookmark. In addition, the system automatically records the bookmarked URL's title and time when the bookmark was created. Delicious's user interface then makes it easy to browse bookmarks along one of three axes: users, URLs, and tags. For example, one can search for all bookmarks created for a given URL, and seem them displayed in reverse-chronological order along with each bookmark's creator, a list of all the tags the user has attached to the bookmark, and an optional brief note. Clicking on a user's name in this list shows all bookmarks created by that user; similarly, clicking on a tag lists all bookmarks "tagged" with that word.

Thanks to this intuitive interface, it is easy for a user to start exploring a given topic area and find out URLs in it or other users with that interest. The social "feel" of Delicious is further enhanced by making it simple to define and view networks of users with shared interests, friends or family, and to share one's bookmarks with people in this network (the "Links for you" feature). Furthermore, to encourage users to stay active and keep up to date, Delicious provides features for monitoring creation of new bookmarks – again, filtered by their creator, tag, or URL, to focus on one's area of interest – either with RSS feeds or "subscriptions" directly from the system.

Delicious also offers several ways to get a more general overview of user activity in the system. The home page shows the "hotlist", URLs that have been bookmarked by large numbers of people very recently, and "tags to watch", a list of most frequently used tags along with most-bookmarked URLs for each tag. The "recent" and "popular" pages expand on the hotlist, showing the chronological and frequency overview of bookmarking activity,

3. FIREFOX AND EXTENSIONS

Mozilla Firefox is an open source web browser developed by the Mozilla Corporation and hundreds of volunteers. The project started in 1998, when Netscape released its commercial web browser Netscape Communicator under an open source license. Firefox is one of the largest and most visible open-source projects today with over 200 million downloads in the last two years alone. However, the project gained momentum slowly, and there was widespread concern in its early years that it is too big and too complex a product to be developed in an open-source manner. This complexity required at least one major rewrite of the code-base, but arguably even more important was the decision in late 2002 to shift the development effort from a "web suite" that incorporated a wide range of functionality to developing stand-alone applications focused on a single-task: the browser (Firefox) and the email client (Thunderbird). The shift was more fundamental than simply changing how applications were packaged and distributed; instead, the project switched its mentality from an "everything but the kitchen sink" outlook to "if it's not essential, make it an extension." This allowed the code to be simplified, and the core developers to focus their effort more effectively. Conversely, the team was now committed to an extensible and fully-documented architecture, so that anyone could write extensions that transparently hooked into the application and could add any new functionality. The result was a success not only on the basis of Firefox's meteoric rise in the market share, but also on the growth of a vibrant ecosystem of third-party extensions, with over 1,800 listed on the official Mozilla extension site.[6]

4. BOOKMAKING FIREFOX EXTENSIONS

Our analysis was performed on a set of Delicious data collected in early September 2006. We started off by retrieving all bookmarks with tags "Firefox", "extension" (or "extensions"), and one of "development", "dev", "programming", "api", "xul",[7] "xpi",[8] "tutorial", "howto", or "tips". This set comprises a total of 2922 bookmarks, identifying 1675 unique URLs.

We then took a sample of 100 bookmarks from this set, comprising 93 unique URLs, and collected the complete history of their URLs. Because of problems with the Delicious retrieval interface, we could only retrieve bookmarking activity for 76 URLs. A total of 24479 bookmarks were created by 17,469 unique users. Within these bookmarks, 3,603 unique tags were used, 83,164 in total. Mean number of tags per bookmark was 3.4, median 3 (range 1–117). Table 1 below presents the descriptive statistics of bookmarking and tagging activity per each URL in the sample:

	Mean	Median	Min	Max
Bookmarks/URL	322.1	68.5	1	2919
Unique tags/URL	110.6	46	3	735
Total tags/URL	1094.3	181	3	9910

Table 1: Statistics for activity per each URL in the sample

5. RELATED WORK

Golder and Huberman have analyzed the structure and dynamics of tagging in Delicious. [2] Looking at the bookmarking of a sample of URLs that appeared in the Popular list in a one week period, as well as a closer look at all bookmarks of a random sample of 200 users, they discovered regularities in user activity, tag frequencies, kinds of tags used, and bursts of popularity in bookmarking. They also discovered that the relative frequencies of tags within a given URL quickly converge to a stable proportion. This behaviour, explained by a simple stochastic model, has important implications for the usefulness of individual tagging behaviour to the wider community, because it shows that a consensus on tags relevant to a given URL forms relatively quickly (after fewer than 100 bookmarks) and remains stable even as other users continue to add their bookmarks and tags for the same URL. Furthermore, the model shows how idiosyncratic, personally-oriented tags that a user may create purely for him- or herself, can coexist with the more general, and commonly-used, tags. Based on these findings,

[5] `http://del.icio.us`

[6] `addons.mozilla.org/search.php?app=firefox&appfilter=firefox&type=E`

[7] XUL is an XML-based markup language used to define user interface elements in Firefox.

[8] XPI is the packaging format for Firefox extensions.

Golder and Huberman argue that while there is no direct evidence for knowledge in the bookmarking data, it certainly seems to occur based on trends in tagging dynamics such as the forming of consensus on tagging vocabulary.

The concept of social bookmarking is being adapted to narrower audiences, where it can be designed to their idiosyncracies and support specific tasks and requirements. For example, Millen *et al.* described an “enterprise” social bookmarking system, adapting the most successful features of a general social bookmarking system like Delicious to use within a large organization. Their system, called “dogear,” links in with other corporate databases and collaboration tools, includes the support for organizational roles, and uses information retrieval techniques to improve detection of users with shared interests and disseminate this information through the system.

Storey *et al.* applied social bookmarking concepts to software development to support cordination and communication in geographically distributed software development teams. [5] Their “tagSEA” tool interprets tags embedded in the source code to create “waypoints,” or landmarks in the code of interest to developers. Waypoints can be viewed as a list organized by tag or location, similarly to bookmarks in Delicious, or can be linked into a “route” – a sequence of landmarks that serves as a guide through a particular portion of the code. Since the source code is shared among the members of the team through a revision control system like CVS, tags, waypoints, and routes are also shared and collaboratively created simply by editing the relevant source files.

6. CONCLUSION

While much more work remains to be done to analyze the bookmarking activity data that we have collected, preliminary results show that Delicious is a rich source of links to information on developing Firefox extensions. The number of users creating bookmarks on this topic shows that there is an audience for tools that help keep track of such information, and that it would be valuable to investigate what additional features would be appropriate in this niche. In the future, we also intend to investigate the dynamics of tagging in this community, as well as extend the collection to other communities, such as Eclipse plug-ins.

7. REFERENCES

[1] K. Crowston and J. Howison. Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology & Policy*, 18(4):65–85, 2006.

[2] S. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2):198–208, 2006.

[3] T. Hammond, T. Hannay, B. Lund, and J. Scott. Social bookmarking tools: A general review. *D-Lib Magazine*, 11(4), Apr. 2005.

[4] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[5] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Waypointing and social tagging to support program navigation. In *Proceedings of ACM SIGCHI conference (CHI’06)*, pages 1367–1372, 2006.

Continuous Coordination (CC): A New Collaboration Paradigm

Ban Al-Ani, Anita Sarma, Gerald Bortis, Isabella Almeida da Silva, Erik Trainer,
André van der Hoek, David Redmiles

University of California, Irvine
Department of Informatics
444 Computer Science Building
Irvine, CA 92697-3440 USA
Phone: +1(949) 824-2776
{balani, asarma, gbortis, ialmeida, etrainer, andre, redmiles}@ics.uci.edu

ABSTRACT

The increase in software complexity introduced the need for software development teams and consequently the need to coordinate team members' activities and create a shared awareness. We seek to overcome some the pitfalls of earlier attempts to coordinate software development through a new coordination paradigm we term *Continuous Coordination* (CC). Generally speaking, the CC paradigm complements formal synchronization with support for informal activities. In this paper, we define the CC paradigm within three dimensions and demonstrate how we embodied CC through a spectrum of Eclipse plug-ins.

CATEGORIES AND SUBJECT DESCRIPTORS

H.4.3 [**Information Systems Applications**]: Office Automation-Groupware; H5.3 [**Information Interfaces and Presentation**] Group and Organization Interfaces – Computer Supported Cooperative Work.

GENERAL TERMS

Management, Design, Human Factors

KEYWORDS

Software Engineer, collaborative work, visualization, configuration management, programming, design.

1. INTRODUCTION

Creating software is an inherently complex task because of its changeable and intangible nature. It is further complicated by the dependencies that exist among artifacts and the gamut of rich interactions required among developers. Distributed software development only adds to this plethora of complexities and further emphasized the need for development environments that provide comprehensive support for different aspects of software development (e.g. Curtis et al., 1988).

The proposed paradigm, Continuous Coordination (CC), blends the best aspects of the more formal, process-oriented approach with those of the more informal, awareness-based approach. In doing so, continuous coordination blends processes to guide users in their day-to-day high-level activities with extensive information sharing and presentation to inform users of relevant, parallel ongoing activities. Thus it provides the underlying infrastructure for coordination. Some of the key properties, we identified, for tools that follow this paradigm are that the tools share *relevant* information and do so in a *contextualized* and *unobtrusive* manner. We deem information *relevant* when it is provided to a developer who will utilize it in the foreseeable future. Shared information is *contextualized* and *unobtrusive* when it is embedded in the development environment allowing developers to modify their behavior at a time that is convenient to them.

Other general tool properties are also being explored in our endeavor to increase the effectiveness of the tools developed within the dimensions of CC. For example, we are of the opinion that developers can need differing levels of information abstraction at various stages of development while carrying out different developmental tasks. We sought to develop a range of tools that can offer a spectrum of support. The tools can then be incorporated into different phases of development by developers as they see fit. Thereby increasing flexibility and providing support for developers' low level programming activities through to high level support of managerial activities.

In this paper, we present a definition of Continuous Coordination dimensions and an outline of some of the tools we have developed thus far within these dimensions. They are discussed in terms of the kind of information it provides and to whom.

2. CONTINUOUS COORDINATION

The CC project sought to address a wide range of needs that are typically manifested during the software engineering process when conducted by co-located or distributed teams. Shared awareness, through shared information is one such need. It has been recognized as being both important and challenging (de Souza et al, 2004).

The challenge in sharing information in this way is achieving an appropriate level of detail and providing it at a time that is suitable to the developers. *How* much information should we provide the developer? Providing a constant stream of information can lead the developers to feel overwhelmed whereas infrequent sharing of information can mean that a developer lacks sufficient information to successfully complete a task.

The information provided to the developer depends on his/her role within the team. *What* kind of information does the developer need? For example, a manager would typically need to be aware of

Conference'06, Month 1–2, 2006, City, State, Country.

team structure, work products and interactions. A programmer, however, would generally need to be aware of changes to the design or code made by other team members.

Finally, we also found that while identifying the amount and the type of information needed by the developers must be determined, the manner in which it is presented should also be considered. *When* should information be shared? For example, if a developer chooses to ignore the shared information this should not impede completing the task at hand. Furthermore, shared information should not distract a developer from the task at hand. The information should be available, such that, a developer can access it at a time suitable to him/her becoming part of his/her *peripheral awareness* in the meanwhile.

In summary, the type of information needed by the developer, the triggers to share information and the recipients of shared information form the three principal CC dimensions. Our approach to embodying the CC paradigm within these dimensions will be discussed in the following section.

3. PLUG-INS: EMBODYING CONTINUOUS COORDINATION

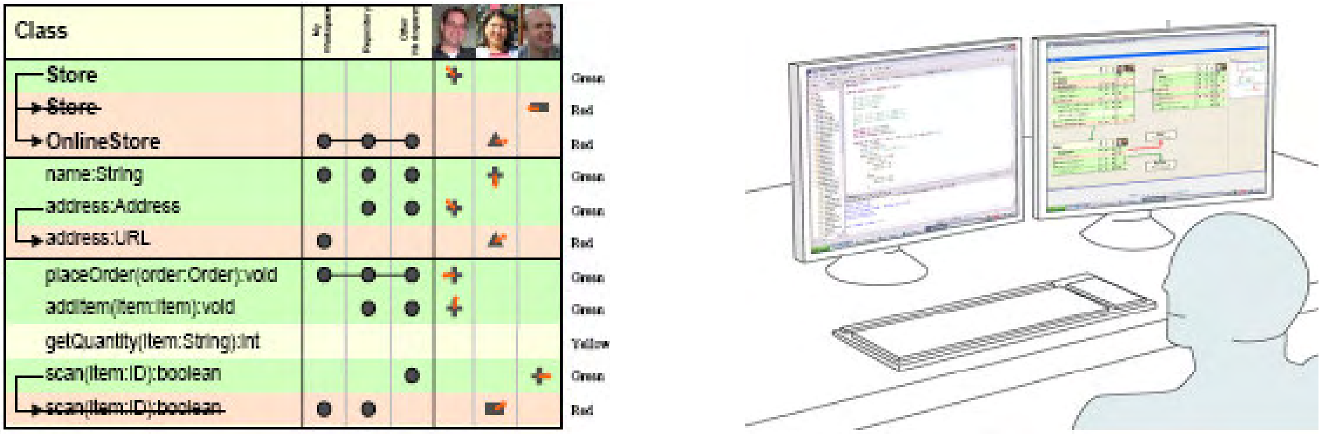
We sought to embody the CC paradigm through a series of Eclipse plug-ins, for several reasons. First, we sought to enable developers to incorporate the proposed plug-ins in a manner suited to the process they have chosen to adopt. In adopting this approach we sought to increase the paradigm's flexibility. Second, we sought to tailor information to individual developer needs and consequently the role they play within the project. We decomposed information that is generally shared, such that, it is possible to identify who it would benefit (e.g. programmers, designer, managers...etc) and potentially minimize the amount of redundant information shared. The third and final reason we embodied CC through plug-ins is because we endeavored to promote self-coordination. While a developer may not be able to coordinate the whole project within the restrictions of their assigned roles they are able, through the proposed series of plug-ins, to coordinate tasks with their peers and those who share their artifacts. Furthermore, some of the plug-ins provide a high level of abstraction or visualizations that can be useful to all types of developers e.g. programmers, designers, managers…etc. In such instances, in this report, they will be referred to collectively as developers only.

3.1 Lighthouse

Lighthouse is a coordination platform that is rooted in the concept of emerging design, a real-time representation of the design as it is being implemented in the code by each of the programmers. Lighthouse then projects this emerging design view on top of the initially conceived conceptual design (da Silva et al, 2006).

Figure 1 illustrates how programmers are able to maintain peripheral awareness of ongoing changes made to the project by the team members when adopting the proposed dual-monitor setup. In this set-up a main monitor would have their primary coding environment and an auxiliary monitor would be dedicated to Lighthouse.

Lighthouse development efforts are currently focused on further improving the user interface such that the changes made to the program is reflected in the design more effectively. Once this is stage is concluded the tool will be validated empirically.



1. a. Project management view.

1. b. Programmer's side-by-side view of code and emerging design.

Figure 1. The emerging design overlaid on top of the original conceptual design produced by Lighthouse.

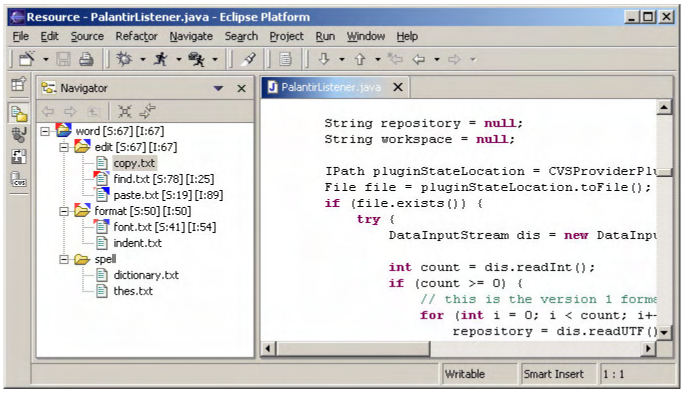
3.2 Palantír

The Palantír plug-in is a workspace awareness tool that provides developers with insight into ongoing development activities in remote workspaces (Sarma et al, 2003). Specifically, Palantír provides information that includes identifying who is conducting a change, what is being changed, calculates a measure of the magnitude of those changes, a measure of the impact of those changes, and graphically displays this information in a configurable and non-obtrusive manner to developers involved in programming (Figure 2).

Palantír breaks the isolation of distributed Configuration Management (CM) workspaces by continuously sharing information of ongoing changes, thereby allowing early detection of conflicts while changes are still in progress. In addition to information regarding which artifacts are being changed by which developer, Palantír distinguishes itself by providing information about the severity and impact of changes. These measures allow developers to gauge which changes are important and require their attention.

Finally, Palantír promotes a model of self-coordination recognizing that many possible and flexible resolutions are possible bringing to distributed development a level of awareness that begins to approach that of local settings. Thus, while the developers are notified of changes, the notification does not impede their work or force them to take immediate action.

Currently, we are in the process of evaluating the effectiveness of Palantír in enabling developers detect potential conflicts earlier and in producing better quality software (i.e. fewer unresolved conflicts) through controlled lab experiments and results are being analyzed.

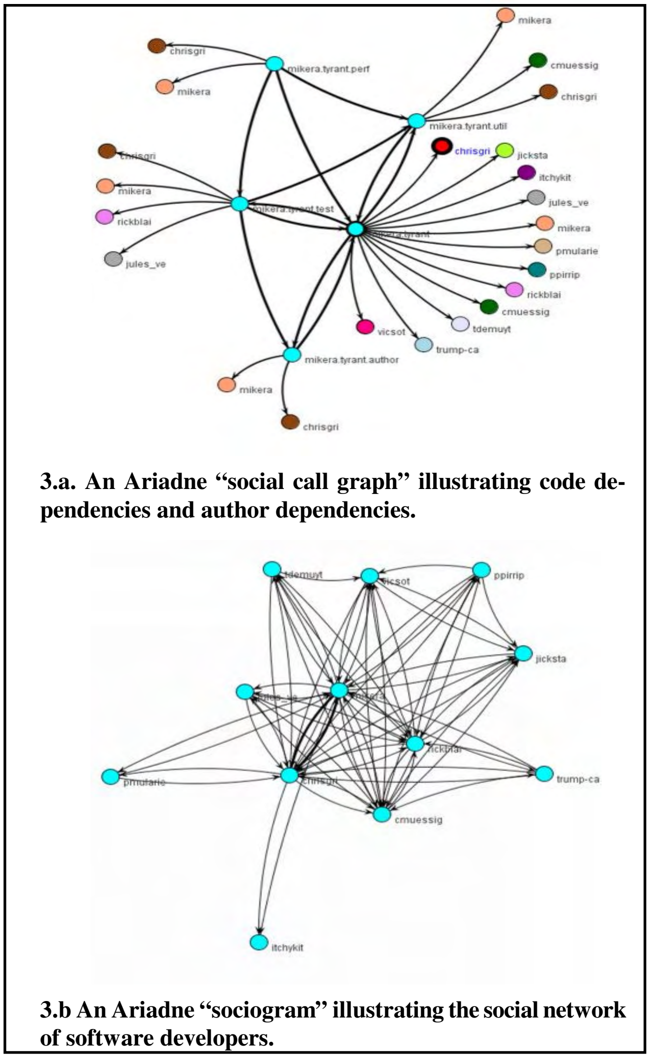


```
String repository = null;
String workspace = null;

IPath pluginStateLocation = CVSProviderPlu
File file = pluginStateLocation.toFile();
if (file.exists()) {
    try {
        DataInputStream dis = new DataInpu

        int count = dis.readInt();
        if (count >= 0) {
            // this is the version 1 forma
            for (int i = 0; i < count; i+-
                repository = dis.readUTF()
```

Figure 2. Palantír Visualization.

3.3 Ariadne

Ariadne is a collaborative software engineering tool that aims to enhance developers' awareness of the social dependencies present in their work by seamlessly integrating such information with development activities (Trainer et al, 2005).



3.a. An Ariadne "social call graph" illustrating code dependencies and author dependencies.

3.b An Ariadne "sociogram" illustrating the social network of software developers.

Figure 3. Examples of graphical representations of social dependencies produced by the Ariadne plug-in.

It analyzes software development projects for source-code dependencies and collects authorship information for the source-code from a configuration management repository. The tool then links the source-code dependencies and authorship information to create a social network of software developers (Figure 3.a). We aim to complement this social network graph with current social network analysis techniques, giving programmers and designers an insight into how their work affects other developers and how the work of other developers affects their own. For example, "centrality" is a measure of the power of nodes as a function of their degree of connectedness with other nodes, their closeness to other nodes in the graph, and their positions as intermediaries between other nodes (Figure 3.b). .

We intend to determine the validity of applying information gleaned from social network metrics to enhance programmers' and designers' awareness of their colleagues' development efforts. Centrality offers a measurement of control or ownership of code, and may help developers identify important players in the project team. Equivalence may be used to help developers identify who is using code similarly to prevent duplication of work.

Finally, Ariadne is currently being trialed to visualize the social interaction within the Ariadne project itself. It succeeded in representing these interaction and the relationships between developers involved. However, initial trials also revealed issues relating to the display of textual information (node labels) and readability. These issues and others will be addressed before conducting extensive empirical studies.

3.4 Dashboard

This plug-in is currently in the design phase of development. A "Wizard of Oz" prototype is being utilized to determine what information would support the social interactions (Figure 4). Ultimately we seek to provide a means to simulate informal "watercooler" conversation by providing a central location where project developers can (1) be informed of their work and how it relates to the overall project, and (2) spontaneously engage in exploration of particular issues raised on the board. Insights gained in this phase of development will assist in developing the first working prototype and determining which visualizations are incorporated in the final product.

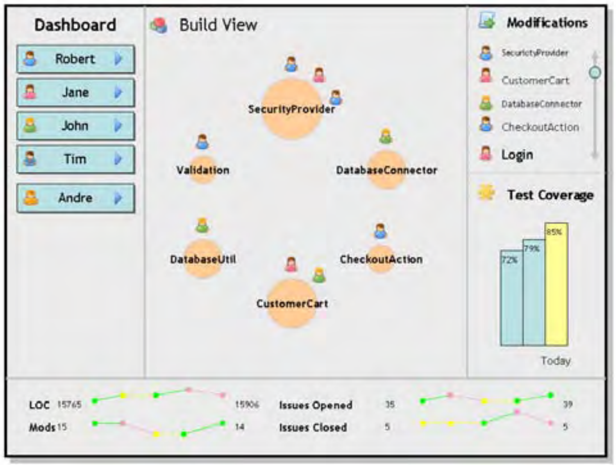


Figure 4. Dashboard provides an abstract view demonstrating the relationship between the developer and the modules under development.

In Figure 4, the programmer's attention is naturally drawn to the spheres, which represent modules that have caused the build to fail. Larger spheres emphasize module importance based on severity metrics. Circling the modules in a clockwise fashion are the names of the programmers who have most recently modified the module. This visual connection between the programmer and the module is further explored in the modifications view (made accessible through the right panel), which displays in real-time the most recent transactions to the configuration management system. Again, more severe modifications are emphasized by a larger module name and sphere.

3.5 World View

World View plug-in provides a comprehensive view of the team dynamics of a project, regarding the geographical location of teams, the time zones of their operations, and the interdependencies among teams (Figure 5). This view is intended to help developers involved in global software identify global and local team members, interactions between sub-groups and other vital information like how to contact global member and when (Sarma and van der Hoek, 2006).

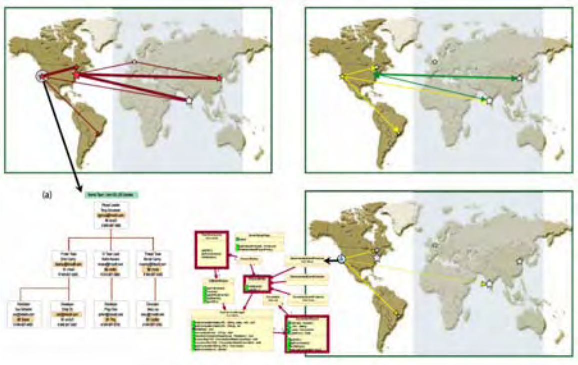


Figure 5. World View screen prototypes. Shaded areas of the map represent countries where it is dark. Active teams in the shaded areas are shown as "white stars".

In Figure 5 teams are represented as "stars" on a world map and interdependencies among teams are shown as "lines" connecting them. The size of the star denotes the size of the team; larger teams are represented as larger stars. Interdependencies among teams are determined based on the number of shared artifacts, which are identified through program analysis of the code base. The thickness of the lines represents the extent of sharing: the thicker the lines, the larger the number of shared artifacts. Through this view, developers can discern at-a-glance which teams are tightly-coupled and through which artifacts (mouse-hovers display the list of shared artifacts).

The directed lines (arrows) in Figure 5 represent the direction of conflicts (changes performed by which team affects which team) and the thickness of the lines denotes the extent of the conflict: the thicker the line, the larger the significance of the conflict. Here, significance is calculated as the number of artifacts that are affected by the change. Teams and their respective "arrows" are color coded to differentiate conflicts arising from different teams. This view can also be configured for the individual developer to show which changes by a specific developer affects other teams. The artifacts responsible for the conflicts are highlighted in red.

Currently, the World View tool is in the exploratory phase with the first prototype to be made available soon.

4. CONCLUDING REMARKS

A new software engineering paradigm was presented in this report, namely: continuous coordination. The project is implemented through a collection of plug-ins. A brief description of each plug in and an outline of the user interface were presented for each. The descriptions sought to demonstrate how each tool fell within the boundaries of the CC dimensions, namely:

1. *Amount of shared information:* each tool provides layers of information such that the developer can adjust the volume and level of detail based on individual need.
2. *Nature of shared information:* the varying forms of information provided by each tool make it possible to focus on information relevant to the task at hand.
3. *Peripheral Awareness:* the information provided by each tool is readily available for the developer to access but does not prevent the developer from continuing with his/her task. The information provided by each tool thus remains within the peripheral awareness of the developers and does not impede their work.

Collectively, these three dimensions seek to define the essence of CC by enabling the developers to *share* an *awareness* of activities carried out during a collaborative development process; such that, developers are neither constrained by the lack of information nor is their work blurred by a high volume of information.

The degree of success achieved by each tool in conforming to these dimensions is yet to be determined through empirical evaluations. However, feedback received from walkthroughs of early prototypes and mock-ups has been positive overall.

5. ACKNOWLEDGEMENTS

This research was supported by the U.S. National Science Foundation under grants 0534775, 0326105, 0093489, and 0205724, by the Intel Corporation, by two IBM Eclipse Technology Exchange grants, and an IBM Technology Fellowship.

6. REFERENCES

[1] Canfora, G.; Cerulo, L., Jimpa: An Eclipse Plug-in for Impact Analysis, *Conference on Software Maintenance and Reengineering (CSMR'06)*, (March 22 - 24, 2006), 341-342.

[2] Curtis, B., H. Krasner, Iscoe, N. (1988). "A field study of the software design process for large systems." Communications of the ACM, 31(11): 1268-1287.

[3] de Souza, C. R., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. 2004. Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work* (Chicago, Illinois, USA, November 06 - 10, 2004).

[4] da Silva, I.A., Chen, P., Van der Westhuizen, C., Ripley, R. and van der Hoek, A. Lighthouse: Coordination through Emerging Design, *OOPSLA Eclipse Technology Exchange Workshop*, October 2006 (to appear).

[5] Sarma, A., Noroozi, Z., and van der Hoek , A., "Palantír: Raising Awareness among Configuration Management Workspaces", *In Proceedings of Twenty-Fifth International Conference on Software Engineering*, p. 444-454, Portland, Oregon (May 2003).

[6] Sarma, A. and A. van der Hoek, "Towards Awareness in the Large", *First International Conference on Global Software Engineering*, Brazil, to appear (October 2006).

[7] Trainer, E., Quirk, S., de Souza, C. R. B., Redmiles, David F. Bridging the Gap between Technical and Social Dependencies with Ariadne. In: Proceedings of the Eclipse Technology eXchange (ETX) Workshop, San Diego, CA, 2005.

The Potential of Instant Messaging for Informal Collaboration in Large-Scale Software Development

Birgit R. Krogstie
Dept. of Computer and Information
Science, Norwegian University of
Science and Technology
Sem Sælands vei 7-9

NO-7491 Trondheim, Norway
+47 73596101

birgitkr@idi.ntnu.no

ABSTRACT

In this paper, we use empirical data to argue that instant messaging can be an adequate tool for informal collaboration in large-scale software development projects. We have conducted a field study on lightweight collaboration tool usage in customer-driven software development student projects. Findings indicate that instant messaging tools can be used successfully for a variety of informal collaboration purposes within a team or small, informal network. We argue that from a theoretical perspective, the usage of collaboration tools in student projects can be seen as relevant to software development work in general, including large-scale projects. There is a lack of empirical research in this area, and we accordingly suggest some directions for future work.

Keywords

Software development, informal collaboration, lightweight collaboration tools, instant messaging

1. INTRODUCTION

"But in the case I am to, have a technical question, and want to ask somebody in the group, I first look at MSN."

(Member of a SD student project team)

Software Development (SD) is a socio-technical activity and inherently complex [1, 10]. Participants need to undertake frequent re-planning and negotiation [7, 17, 18] with reference to a shared workspace. When radical collocation is not possible, the need for coordination increases [1, 11].

A large part of SD project work amounts to programming-related tasks. The need to support the more "fine-grained", code-related aspects of software development has recently been addressed by [4]. Gutwin and colleagues have addressed how distributed developers in open source projects solve their needs for awareness of the team. They found that this was done mainly through text-based communication in the form of mailing lists and text chat [10]. Functionality to support informal collaboration in SD may be integrated into collaborative development environments. Alternatively, lightweight collaboration tools may be used in parallel with development tools.

Instant messaging (IM) is a lightweight tool seen in regular use in work life [2, 15, 19]. It offers opportunities for easily initiated, synchronous and partly asynchronous chat with one or more members of a buddy list in which brief status information about each member is displayed (e.g. 'away'). Other features typically included are the option to set up individual and shared spaces as well as file transfer between users. Whereas the functionality provided by IM tools may thus in principle be adequate for supporting informal collaboration in SD work, the possibility of using IM in SD has received little attention in the CSCW field.

In this paper, we address IM usage in SD work with reference to empirical data from customer-driven SD student projects, a type of project briefly presented in Section 2. Our cases are presented in Section 3. In Section 4, we outline the research method, and the main findings are presented Section 5. We next discuss limitations to the study, possibilities for generalization of findings to professional, large-scale SD work, and directions for further work, before concluding with a summarized position.

2. CUSTOMER-DRIVEN SOFTWARE DEVELOPMENT STUDENT PROJECTS

Student project represent a particular kind of SD project work, with many elements that correspond well to professional SD work and some elements that belong to a formal education setting.

SE student projects are examples of project based learning proven successful in SD study programmes. The project students typically are in the last phase of their SD education, finalising their bachelor's or master's degrees. The projects normally take up most of one semester, and are generally given high priority by the students. The projects provide learning that cannot be achieved through traditional course formats. *Customer-driven* SE projects have external customers and 'real' problems that add authenticity and a stronger aspect of situated learning.

There are three main stakeholders in a customer-driven SE student project: A group of students forming the project team, the customer (usually an organization external to the university,) with a real-life problem to be solved by the group, and the university staff associated with the course. Normally, a documented software product is to be developed for the customer. The university additionally requires documents reporting on project management and the students' reflection on the process, and a final report is an important basis for evaluation of the group's work.

3. CASES

We have collected data from two, in many ways similar, bachelor level, customer-driven SD project courses in Norway. Project Course 1 is offered by a private university college offering study programs within IT. This is the former workplace of the author. Project Course 2 is part of the IT bachelor program at a university, the current workplace of the author.

The student projects are conducted in groups of 3-5 students, who receive a common grade on their result. Whereas both project courses are customer-driven, Project Course 2 has a bigger share of university-internal customers. Project Course 1 generally has college-external customers. In Project Course 1, there is a requirement that the customer offers adequately equipped office space for the students three days a week during the course of the project, and the students are expected to spend most of their project working time there. In Project Course 2, students occasionally visit their customers for meetings, but mainly stay at university campus (or at home) while working on their project.

4. RESEARCH METHOD

We have conducted field research on the two SD project courses, collecting qualitative data from various sources. The original research objective was to investigate mobility in SE student groups, understood an issue not only of physical/geographical movement, but also of switching between different collaboration contexts. Also, we intended to investigate the groups' use of collaboration tools to support mobility, with the aim of suggesting new or improved technologies to be tried out in similar settings.

Initially, focus was on Project Course 1, in which one project group was observed (and video recorded) on several occasions during their project. The objective was to capture relevant aspects of their work in different collaboration contexts: within the team only, with the supervisor, and with the customer. We asked the group to take a photo whenever there was a new collaboration setting (e.g. new technology, new people, new location), and post it to a blog. This provided us with information about relevant situations to observe. Based on the observations, an interview guide for a semi-structured interview across groups was designed, addressing challenges regarding collaboration support in their situation of distributed and – arguably – mobile work. 7 out of the 11 groups in the course agreed to participate in an audio recorded 1 hour interview. The interviews were held about 3 weeks before the project deadline. The recordings were partly transcribed, partly summarized in detail, and next coded in accordance with a combination of initial categories and a grounded approach. The analysis revealed patterns in the use of lightweight collaboration tools that were in themselves interesting and that indicated promising directions for further empirical, longitudinal research.

The data from Project Course 1 on the use of lightweight tools mainly shed light on *within-team* use of these tools. In Project Course 2, supervised by the author as a member of the course staff, we found an example of a project group using MSN to collaborate with their customer abroad. The group logged their conversations to document their project process. After having participants' consent to use the material, made anonymous, for research purposes, we included the MSN logs in our data material illuminating use of lightweight collaboration technology in SD.

In addition to the data described, we have, from both courses, access to project grades, students' reflection notes from the projects, and customers' views on their objectives for and outcomes from the projects (as described to us in brief telephone interviews conducted in August 2006, retrospectively to the project). These data are not used in the context of this paper.

5. FINDINGS

The Project Course 1 students appear very conscious about which tool they use for collaboration under different circumstances. Generally, they present a clear hierarchy of preferred technologies for initiating informal collaboration when working distributedly. *All* the 7 groups have MSN Messenger on top.

Reporting on their use of MSN, the groups interviewed describe *within-team collaboration* only. When they refer to collaboration with the customer or the university, they mention formal or informal face-to-face meetings, email, and telephone, and sometimes virtual meetings or document/feedback exchange via shared project workspace (depending on the infrastructure offered and/or preferred by the customer), but they *never* mention IM.

Groups express themselves in a way demonstrating how the use of MSN and "being on" within and outside work hours is taken for granted. If the group is distributed, availability for synchronous, project-related communication is determined by the status indicated in the MSN buddy list. The list may thus be seen as an important source of presence information about team members.

The buddy list is very long in some cases, and agreed-upon protocols may be needed to filter communication. For instance, one group reported that the message 'away' by someone's name was to be interpreted as 'probably here; will answer messages from high-priority contacts (e.g., SD team members) only'.

MSN is often used on the side of other work tasks, typically programming, and serves as a means for fine-grained coordination and awareness about, and access to, the shared object of work. For instance, programmers may each have a terminal window to a shared server while discussing about what happens as they make changes to the code and build and deploy the system. Major project decisions e.g. regarding the overall system design are typically referred to face-to-face meetings. MSN is also used for distributing material, e.g. screenshots or pieces of source code.

Generally, work-related MSN chat is done among two participants at the time. Exceptions are made at need: One group reported that during Easter (mainly holiday in Norway), they had a chat with several participants because everybody was working on the same task, and they had some problems making it work. In parallel to the MSN chat on this occasion, the group members used a server logon to enable everybody to see what was happening on the server, on which something was shown by the programmers.

Some groups reflected over the shortcomings of MSN as compared to the advantages of collocated work. The possibility of misunderstandings due to the more brief form of expression than in normally conversation was mentioned, and some found it difficult to discuss revisions of a document via MSN.

The use of IM in the project groups interviewed is restricted to PCs. Without exception in our data, in the context of project work, the students only use their mobile phones – which are plain and inexpensive - for speaking and exchanging SMS.

An aspect of the project work generally mentioned among the teams is the significance for collaboration of the group members *knowing each other in advance*. The students in each group generally knew each other because they had formed groups themselves, typically based on previous cooperation. When asked about success factors for (student) SD projects, the groups mention knowing each others' competencies, being aware of each others weak and strong sides, avoiding the initial phases of group process, and not having to explain everything.

About the use of IM as a collaboration tool between project group and customer, we have limited empirical data. The logged MSN conversations taken from Project Course 2 have not yet been fully analysed, having only recently become available to us. The logs show that the students, communicating with a customer who is situated on the other side of the globe and whom they have never met, try to keep an informal tone while discussing issues crucial to the success of the project, e.g. functional requirements and server access. There is a breakdown in conversation, the customer abruptly leaving after some rather direct statements from the project group. (The author of this paper, as supervisor of the group, interfered after this conversation took place, writing the customer an email to re-establish the ground for collaboration between customer and group, which partly succeeded.)

The logs document an *atypical* case compared to the other projects, which is why we choose to include it in our data. IM is tried out in the context of project work – with mixed success. Theory on the adequacy of different communication media under different circumstances and for different purposes may account for the breakdown in communication in this case. Further analysis of the log data, supplemented with follow-up interviews of the parties involved, will be an important source of an explanation, which is likely to have several components, e.g.: The parties did not know each other personally; cultural differences; lack of understanding of the customer's needs and requirements, of the students' competence and/or expectations, and/or the goals or pedagogical design of the project course; the participants were trying to use medium fit for informal collaboration in a situation of formal collaboration, and/or trying to make collaboration informal when in fact serious, unclarified project issues were discussed. Lack of media richness might have been the problem, or the parties might have had an attitude which would have made collaboration difficult over any medium.

6. DISCUSSION

To generalize from SD student projects to large-scale SD work, we need to account for two dimensions (see Figure 1): Going from an educational setting for project work to a professional setting, and from small-scale to large-scale SD work.

Any specific SD project is of course subject to particular conditions, e.g. in terms of size, criticality, methodologies, policies, distribution of work etc., but we simplify the picture and suggest aspects of student SD projects that resemble professional SD work, and aspects that have less obvious counterparts there.

Customer-driven student SD projects resemble professional SD work in having a customer with requirements for a product to be developed, documented and actually taken into use; being organized as a project with requirements for proper project management in accordance with a process model; being conducted by a team with joint responsibility for the result (at least typical for Norwegian student projects); and having a set of development and collaboration tools available for the developers.

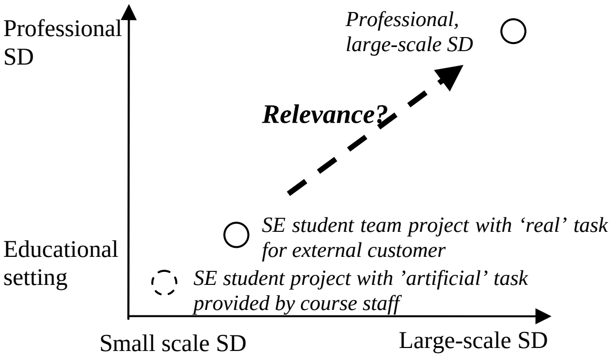

Professional SD
Educational setting
Small scale SD
Large-scale SD
Professional, large-scale SD
Relevance?
SE student team project with 'real' task for external customer
SE student project with 'artificial' task provided by course staff

Figure 1: From student projects to large-scale SD

Education-related aspects of SD student projects include: There is a pedagogical design with learning objectives for the course, and these objectives typically reflect a relationship with a number of other course modules in the study program. The project is subject to formal evaluation according to a set of criteria covering both product quality (e.g. architecture, design, usability) and the quality of the project process (e.g. work breakdown, estimation, follow-up of plans, resolution of conflicts, post-hoc reflection). There is supervision from university staff. Reports are required by the university, usually including more extensive documentation of the project than is required by the customer. The need to document process and communication from a legal point of view, is small. There is little economic risk associated with the project: the developers are not paid for their work, and the customer contributes only with a limited amount of resources (e.g. office space and some work hours for supervision). For the members of the development team, there is limited personal/career risk associated with failure. (The final grade for the project is however important to most students.) The project is small-size. Students generally lack expensive technical equipment for private use.

Many of the above 'education aspects' of student SD projects may however have counterparts in professional SD, e.g. due to a focus on organizational learning and knowledge management.

Developers' personal habits of lightweight tool use may be important in generalizing from student projects to professional SD. Lightweight tools such as SMS and IM tools have come to play an important role in the life of teenagers as a means for developing and maintaining peer group membership and social identity [8, 9, 14]. Young users are conscious about how IM supports both social and work-related interaction and what features of IM are desirable for different purposes [12]. The tools, and the links they contain to other members of the social network, may be seen as part of a *personal communication infrastructure*, used for the maintenance of social relations and potentially for informal, work-related collaboration. This infrastructure will change in accordance with the individual's affiliation with a school or workplace, but at the same time it is persistent over time and across occupational affiliation. For instance, the buddy list of an IM tool typically includes both friends and colleagues.

In the context of SD, we argue that when developers embark on their professional careers, it is not only the existing routines, technologies and policies of their employers' organization that will form their actual usage of collaboration tools. Young developers who are active SMS and IM users will have their habits of lightweight collaboration as part of their *formative context* [3] or *technological frame* [16] affecting their work.

When generalizing from small-scale to large-scale SD, a key factor is the informal networks serving as the backbone of organizational interaction [13]. Interaction in these networks comprise affect, politics, production and culture [20], typically among small clusters of people. Programmers use different communication nets for different types of work-related issues [6]. In open source development, interaction is often decentralized [5]. Tools used for informal collaboration in small SD teams may be adequate for similar use within clusters in large-scale SD projects.

In our interviews, we have identified patterns across groups rather than focusing on the particular characteristics of each particular group. Generalizing to similar project courses, we need to be aware that the heavy usage of MSN messenger in the particular population studied is not necessarily representative of all, bachelor level SD students, even in Norway. If IM is in fact particularly popular in our sample project course, the findings however still demonstrate the potential of IM in project work.

There are limitations to the use of interview as a research method in our context. The data are filtered through students' perception. Also, we asked students to report to us in the presence of the rest of their groups, which may have influenced the responses. We gathered data from each group as a unit, which is a simplification. To understand collaboration over time, and to get an impression less flavored by participants' view on their own process, some form of observation over time is required. Also, we have made no follow-up study to clarify issues emerging from the data.

There are many factors that might have contributed to the successful within-team use of IM in our case, for instance: The projects are small in scope, the developers belong to a generation having IM as an important part of their personal infrastructure, and the developers know each other before the project starts. We still interpret our findings as indicating that IM can be useful in supporting informal collaboration in SD projects.

As previously discussed, there are many possible reasons why our example of team-customer communication via MSN broke down, and we believe further analysis of the data may provide more answers.

Whether *mobile* use of IM, not identified in our data, will be part of the students' habits in the years to come, remains to be seen, and is an interesting issue for further research.

7. CONCLUSION

Our position is that IM holds a potential to support informal collaboration within clusters of people who know each other in a large-scale SD organization. We suggest that great caution should be made before IM is used whenever communication is of a more formal character, the participants do not know each other, or the issues addressed involve major decisions for the project

8. AKNOWLEDGEMENT

The author wishes to thank Bendik Bygstad for useful insights

9. REFERENCES

[1] Carstensen, P.H. and Schmidt, K. Computer Supported Cooperative Work: New Challenges to Systems Design. in Itoh, K. ed. *Handbook of Human Factors/Ergonomics*, Asakura Publishing, Tokyo, 2002 (1999).

[2] Cherry, S.M. IM means business. *Spectrum, IEEE*, *39* (11). 28 - 32

[3] Ciborra, C.U. and Lanzara, G.F. Formative Contexts and Information Technology: Understanding the Dynamics of Innovation in Organizations. *Accounting, management and information technologies*, *4* (2). 61-86.

[4] Cook, C., Churcher, N. and Irwin, W., Towards Synchronous Collaborative Software Engineering. in *11th Asia-Pacific Software Engineering Conference (APSEC)*, (Busan, Korea, 2004), IEEE Computer Society.

[5] Crowston, K. and Howison, J. The social structure of free and open source software development. *First Monday*.

[6] Curtis, B., Krasner, H. and Iscoe, N. A field study of the software design process for large systems. *Communications of the ACM*, *31* (11).

[7] Farshchian, B. Presence technologies for informal collaboration. in *Emerging Communication: Studies on new technologies and practices in communication*, IOS Press, 2002.

[8] Grinter, R.E. and Eldridge, M., Wan2tlk?: Everyday Text Messaging. in *CHI2003*, (Ft.Lauderdale, Florida, USA, 2003), ACM.

[9] Grinter, R.E. and Palen, L., Instant Messaging in Teen Life. in *CSCW'02*, (New Orelans, Louisiana, USA, 2002), ACM.

[10] Gutwin, C., Penner, R. and Schneider, K., Group Awareness in Distributed Software Development. in *CSCW*, (Chicago, Illinois, USA, 2004), ACM.

[11] Hersleb, J.D., Mockus, A., Finholt, T.A. and Grinter, R., E., An Empirical Study of Global Software Development: Distance and Speed. in *International Conference on Software Engineering*, (Toronto, Ontario, Canada, 2001), IEEE Computer Society, 81-90.

[12] Huang, A.H. and Yen, D.C. Usefulness of instant messaging among young users: Social vs. work perspective. *Human Systems Management*, *22* (2). 63-72.

[13] Krackhardt, D. and Hanson, J.R. Informal networks: the company behind the chart. *Harvard Business Review*, *71* (4). 104-111.

[14] Lewis, C. and Fabos, B. Instant messaging, literacies, and social identities. *Reading Research Quarterly*, *40* (4).

[15] Nardi, B.A., Whittaker, S. and Bradner, E., Interaction and Outeraction: Instant Messaging in Action. in *CSCW'00*, (Philadelphia, PA, USA, 2000), ACM.

[16] Orlikowski, W., Learning from Notes: organisational issues in groupware implementation. in *CSCW*, (Toronto, Canada, 1992), ACM.

[17] Rönkkö, K., Dittrich, Y. and Randall, D. When Plans do not Work Out: How Plans are Used in Software Development Projects. *Computer Supported Cooperative Work 14*.

[18] Schmidt, K. and Bannon, L. Taking CSCW Seriously: Supporting Articulation Work. *Computer Supported Cooperative Work (CSCW): An International Journal*, *1* (1). 7-40.

[19] Scupelli, P., Kiesler, S., Fussell, S.R. and Chen, C., Late Breaking Results: Posters: Project View IM: A Tool for Juggling Multiple Projects and Teams. in *CHI 2005*, (Portland, Oregon, USA, 2005), ACM Press.

[20] Waldstrøm, C. Informal Networks in Organizations - A literature review *S-WOBA (Scandinavian Working Papers in Business Administration)*, Aarhus School of Business, Aarhus, Denmark, 2001.