



ASE2006

21st IEEE/ACM International Conference on Automated Software Engineering

September 18-22, 2006

Workshop



Tokyo, JAPAN



Proceedings of the 2nd International Workshop on
Supporting Knowledge Collaboration in
Software Development
(KCSD2006)

September 19, 2006

Editors: Yunwen Ye & Masao Ohira

ISBN 4-86049-036-3 National Institute of Informatics (Tokyo, Japan)



Proceedings of the

2nd International Workshop on

Supporting Knowledge Collaboration in

Software Development

KCSD2006

September 19, 2006
Tokyo

In conjunction with
21st IEEE/ACM International Conference on
Automated Software Engineering (ASE2006)

Editors:
Yunwen Ye & Masao Ohira

Table of Contents

Workshop Organization	v
Session 1: Collaboration and Communication	
Supporting Software Development as Collective Creative Knowledge Work	1
<i>Kumiyo Nakakoji (University of Tokyo, Japan & SRA Key Technology Lab, Japan)</i>	
When Programmers Don't Ask	9
<i>Sukanya Ratanotayanon, Susan Elliott Sim (University of California, Irvine, USA)</i>	
Session 2: Knowledge Access	
Recommending Library Methods: An Evaluation of Bayesian Network Classifiers	17
<i>Frank McCarey, Mel Cinneide, Nicholas Kushmerick (University College Dublin, Ireland)</i>	
Assisting Concept Assignment Using Probabilistic Classification and Cognitive Mapping	25
<i>Brendan Cleary, Chris Exton (University of Limerick, Ireland)</i>	
(Position Paper)	
A Tool-Supported Environment for Knowledge Feedback Cycle in Software Development	33
<i>Noriko Hanakawa (Hannan University, Japan)</i>	
Session 3: Community Knowledge	
Social Network Analysis on Communications for Knowledge Collaboration in OSS Communities.....	35
<i>Takeshi Kakimoto, Yasutaka Kamei, Masao Ohira, Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan)</i>	
The Flow of Knowledge in Free and Open Source Communities	42
<i>Daniel M. German (University of Victoria, Canada)</i>	
(Position Paper)	
Using SNS Systems to Support Knowledge Collaboration	50
<i>Masahiko Ishikawa (Software Research Associates, Inc., Japan)</i>	
Session 4: Project Knowledge	
Building the Knowledge Network in Software Project	52
<i>Atsushi Inuzuka (Japan Advanced Institute of Science and Technology, Japan)</i>	
(Position Paper)	
Coordinating Multi-Team Variability Modeling in Product Line Engineering	60
<i>Deepak Dhungana, Rick Rabiser, Paul Grunbacher (Johannes Kepler University, Austria)</i>	
(Position Paper)	
Learning Support by Reflection and Knowledge Collaboration in a	62
Team-based Software Engineering Project Course <i>Atsuo Hazeyama (Tokyo Gakugei University, Japan)</i>	
(Position Paper)	
Knowledge Collaboration by Mining Software Repositories	64
<i>Thomas Zimmermann (Saarland University, Germany)</i>	

Workshop Organization

Workshop Co-Chairs

Yunwen Ye

University of Colorado USA & SRA Key Technology Laboratory, Japan

Masao Ohira

Nara Institute of Science and Technology, Japan

Program Committee

Gerhard Fischer, University of Colorado, USA

John Grundy, University of Auckland, New Zealand

Katsuro Inoue, Osaka University, Japan

Kouichi Kishida, SRA Key Technology Lab, Japan

Kei Kurakawa, Nara Institute of Science and Technology, Japan

Karim Lakhani, Massachusetts Institute of Technology, USA

Jianguo Lu, University of Windsor, Canada

Ken'ichi Matsumoto, Nara Institute of Science and Technology, Japan

Kumiyo Nakakoji, University of Tokyo, Japan

David Redmiles, University of California, Irvine, USA

Tao Xie, North Carolina State University, USA

Lu Zhang, Beijing University, China

Binyu Zang, Fudan University, China

Supporting Software Development as Collective Creative Knowledge Work

Kumiyo Nakakoji^{1,2}

¹*RCAST, University of Tokyo* ²*SRA Key Technology Laboratory Inc.,
kumiyo@kid.rcast.u-tokyo.ac.jp*

Abstract

We view software development as a system of evolution consisting of the three elements: (1) artifacts, (2) individual developers, and (3) a community of developers. An individual's determining what artifacts to contribute and how, with whom to communicate by asking or answering, and which role to play within the community affects the quality of software to be developed; how the developers relate to each other does matter. Software development should be viewed as a system of evolution driven through metabolic processes of how artifacts, developers, and the community grow. This paper describes the framework of viewing software design as a collective creative knowledge work, and outlines possible research areas to pursue.

1. Introduction

Software development is knowledge intensive work, involving both planning and presentation activities [34]. Developers need to locate source code potentially relevant to the task at hand, understand how to modify the source code while identifying why the way it is, and/or write new code where necessary [20]. Although requirement specifications, design documents, comments, and design rationale are provided to help developers in this process, they are often not enough. Developers need to be familiar with the programming language for the code, component libraries used and potentially usable for implementing the code, design methods applied to develop the code, programming tools and environments available to develop the code, and application domains of the code.

While experience is certainly helpful, it does not necessarily work in such a way that a longer experience of engaging in a development project provides more knowledge about the entire project. Software development needs knowledge in a variety of fields, which require constant updates. There are no absolute *experts* in software development. Application domains are subject to rapid change. Component libraries are continually updated. New features and functionalities keep being introduced in programming tools and environments. Moreover, there is such

culture in software development that keeps 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” [20]. Thus, the “symmetry of ignorance” among a development team is neither a problem nor an accident; it is a matter of fact in software development [8].

This makes software development a fundamentally social activity [30]. The activity is carried out by a group of developers, forming a community, engaging in collective creative knowledge work [26]. It is a social activity mediated through artifacts, which are primarily source codes and documents. Even a single-person project has such a community aspect because the project is likely to use component libraries and existing modules, which have been developed by a number of other developers over a long period of time.

2. Social Aspects of Software Development

Social aspects of software development have been studied mostly in the context of how developers and end-users work together in designing a computer technology. Ethnographers and social scientists have explored ways to help them develop a shared understanding and shared context during the process [41]. Another social aspect that has been studied is the organizational context of a software development project [30].

This paper in contrast focuses on the peer-to-peer level of knowledge collaboration of software developers. How developers use other developers as knowledge resources and what social issues are involved during the process, such as the cost of interruption and the motivation for contribution.

Let me first illustrate what kinds of social issues I am referring to.

While sharing knowledge and information within a community of developers is indispensable, the primary means for developers to obtain knowledge is not

through communicating with their peers, but through artifacts.

In understanding source code, developers ask questions such as where to focus as an initial point, exploring the related parts, understand concepts involved the related parts, and understand the relationships among the concepts [35]. During the process, software developers “invest great effort recovering implicit knowledge by exploring code” [20].

However, this exploration process often does not succeed primarily because of the lack of detailed knowledge articulated in the source code. If this becomes the case, software developers would start depending on distributed knowledge resources; namely, the other developers in the community.

By conducting two surveys and eleven interviews with software developers at Microsoft Corporation, [20] have observed that “Developers go to great lengths to create and maintain rich mental models of code that are rarely permanently recorded, and when trying to understand a piece of code, developers turn first to the code itself but when that fails, to their social network.” This would work because source code is often *owned* by a certain developer or a team of the small number of developers, who has a detailed, almost complete knowledge of the source code.

This way of knowledge sharing and collaboration involves two types of social issues. First, asking the *owner* of the source code, either through face-to-face or via email would cost some additional work for the person who is being asked for help, and may interrupt his/her primary work [20]. An interruption is regarded as an unexpected encounter initiated by another person, which disturbs “the flow and continuity of an individual’s work and brings that work to a temporary halt to the one who is interrupted” [37]. Different interruption moments have different impacts on user emotional state and positive social attribution [1].

Second, even if the one understands the source code by being helped by his/her peer, this understanding is not likely to be articulated nor recorded because not only of the overhead of writing it down, but also of the feeling that the newly found information “is not authoritative enough to add permanently to the code” or that checking in the comment under his/her own name “would inappropriately make them experts” [20]. This thereby often results in “institutional memory loss” [20].

Supporting software developers would need to support their collaboration with their peers. Support for collaboration, then, would need more than simply finding the “right” person for completing the task. Social factors, such as motivation, trust, self-confidence and social recognition, need to be dealt with.

3. Three Elements of Software Development: Artifacts, Developers, and a Community

The goal of supporting software development as collective creative knowledge work is to support software developers to develop software. This is different from that of social matching systems, which is to introduce people to people [38].

This position paper views software development as a system of evolution consisting of the three 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 forming a knowledge community. A knowledge community is a group or people that collaborate with one another for the construction of artifacts of lasting value [4]. In a knowledge community, people are bonded through the construction of artifacts.

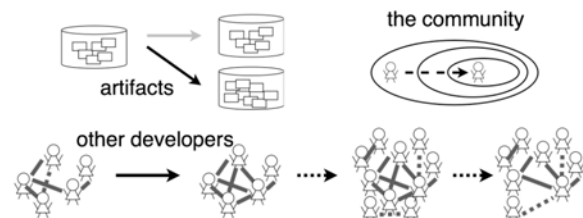


Figure 1: Software Development as a System of Evolution Consisting of the Three Elements

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 the engagement in the project. Such factors affect how the developers collaborate, communicate, and coordinate with one another, resulting in different ways of how they share knowledge.

Because sharing knowledge is indispensable in software development, the quality of resulting software thus depends not only on the skills and knowledge of individual developers but also on the roles of and the social relationships among the developers. The sum of the amount of each developer’s knowledge does not simply determine the quality of software to be developed; how the developers relate to each other also does.

None of the three elements are constant during the software development. Artifacts change over time throughout the development. Individual developers, or more precisely, what individual developers know, grow by gaining experiences of engaging in the development and learning about the artifacts. A

community of developers change by having new developers join, and some developers leave the development project. Their officially assigned roles and informally perceived roles change over time, and the social relationships among them also change.

Existing studies on supporting software development have primarily focused on the evolution of artifacts. More recent work has started to look at how individuals change over time through learning. In contrast, not much has been studied on the aspect of the evolutionary community in the context of software development processes [27].

The rest of the position paper focuses on how the community element evolves and how technologies ought to support such processes.

4. The Metabolic Process of a System of Evolution

A software system needs to evolve to improve its quality in terms of efficiency and robustness, or to cope with the external changes in the environment where the software is used. This type of evolution, recently referred as incremental change [32], should be viewed as not simply adding new objects or mending broken ones; rather it should be viewed as a metabolic process.

Artifacts go through such a metabolic process by adding, modifying, and refactoring the source codes. New parts are added and old parts are rewritten. Some parts may be replaced with other parts.

Individuals' knowledge evolves through learning [22]. They learn by reading source code and information sources such as documents. They learn by asking peers questions. They also learn by solving new problems and experiencing unfamiliar situations. Their old knowledge is replaced with new ones and restructured during the learning process.

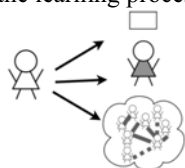


Figure 2: Three Aspects of the Community's Metabolic Process

A community grows through a metabolic process through individual activities. This paper views the metabolic process of a community from the following three aspects (Figure 2): (1) the relationship of an individual with artifacts; (2) the relationship of an individual with other developers; and (3) the relationship of an individual to the community as a whole.

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

An early social navigation system, Expert Browser [24], provides this type of information. Expertise browser uses data from change management systems to locate people with desired expertise, by using a quantification of experience. The system then presents evidence to validate this quantification as a measure of expertise.

As a more recent tool, LifeSource [14] provides two visualizations of CVS code repositories. CodeConnections provides file-centric, temporary-animated visualizations, where color-coded authors (i.e., developers) are indicated in terms of the file-structures. CodeSaw provides the author-centric visualizations of weighted collection of email and code contributions of each developer, where the view can overlay multiple developers' contribution to make comparisons.

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



Figure 3: Asker-Helper Relationship

Answering to a question costs the helper (i.e., the answerer) additional work as well as interrupts the helper's current task. Resuming the original task after such an interruption has also been found quite costly [19]. How one would, then, help another if answering is such a costly task.

To help people to decide to whom to ask, social awareness tools [36] help community members become aware of what is going on within a community, and primarily helps askers to decide who and when to ask a question by looking at how intensively potential helpers are currently engaging in their own tasks.

Comparing to the number of approaches that aim at supporting askers, very few studies exist that focus on supporting helpers (Figure 3).

The feeling of *expectation* and *obligation* plays an important role during the helper's process of deciding whether and when to help. Having information about one's social relationships with the other individual developers helps him/her develop a feeling of *obligation* and *expectation* with each of them [28] as people tend to favor reciprocal acts. If Person X gives a service to Person Y, X feels an expectation for Y while Y feels obliged to return the service to X in the near future. Thus, one may feel obliged to answer to a question being asked by his/her peer developer who had kindly helped him/her the week before. *Obligations* "represent a commitment of duty to undertake some activity in the future" [25]. *Expectations* are what one has on others based on one's trust to them and vice versa. Researchers see obligations and expectations as complementary features [3], incurred during prior interactions, creating value for the community in the future [31].

A few systems have been developed to explore individual relationships to help one decide how to engage in the communication. For instance, Soylent [9] provides temporal and social structures of an online activity by visualizing email messages and their traffics. The system provides a nice ego-centric view to identify with whom one has been communicating at what time, helping him/her to develop a feeling of obligation and expectation.

(3) *the relationship of an individual to the community as a whole*. How one relates with the community is concerned with his/her role within the community, belonging to either a peripheral part, a core part, or an intermediate layer. This aspect helps a developer to decide how much he/she contributes to the community by getting trust and social reputations within the community. One's role evolves within a community through legitimate peripheral participation [40]. By looking at how and what a developer's peers who are closer to the *core* of the community do within the community, he/she gradually acquires skills through learning, and develops his/her identity within the community.

One-to-one communication and collaboration also contributes to the development of social reputation. *Obligations* and *expectations* also play a role in this context. When other peers in the community look at X giving service to Y, X might gain not only expectation to Y but also social reward from the community in the form of good reputation and trust from the community. This might then lead to shifting the role of X within the community from the *periphery* closer to the *core*.

Tools have been developed to use Usenet Newsgroup communities to identify this type of relationships of an individual with the community.

Tools described in [10][12] provide a second-degree ego-centric network for each author together with out-degree histograms of each community, and identify types of users (e.g., answer-only group) and characterizes each community. Newsgroup Crowds and AuthorLines [39] identify authors and types of authors in terms of how they are engaged in the community. The tools visually represent for each user the number of posting per thread and that of active days over a month. They highlight recently posted messages and the encodes the number of posts to the entire set (Usenet Newsgroup as a whole) as the size, allowing people to understand the "role" of a user as a whole and for a particular newsgroup.

In order to support the evolutionary metabolic process of a community, we need technologies for individuals to become aware of the current state as well as its history from the three aspects; that is, to help them determine what artifacts to contribute and how, with whom to communicate by asking or answering, and which role to play within the community.

5. Technical Support for Metabolic Processes of the Community Evolution

Our approach is to use the interaction histories as a source for such decision making by allowing developers to deal with social factors, such as motivation, trust, self-confidence and social recognition.

A number of social navigation systems have been studied to support community activities in a variety of domains [17]. Many of them visualize the history of the community members' activities to analyze the community as a whole, and/or to help a user decide which community to join or to find people with whom the user should communicate. Many of them, however, suffer from not having a clear goal of who is to use the visualizations for what purposes. Having clear goals would determine what types of data to show in what ways, for instance, whether to use *my-own-data* or *collective-social-data* as a *collective snapshot* or as *temporal transitions* [11].

Our goal here is to use the interaction history data to help software developers determine how to engage in the community by interacting with which artifacts and with whom. How developers engage in the community then would shape the metabolic processes of the system of evolution from the community aspect. In considering this, this section argues for the following claims.

(1) *Such data should describe the state of the community, as well as the trends and temporal changes over a long period of time.*

The evolution of an organism depends not only on the type of perturbation, but also on the current structure of the organism. The current structure is determined through its historical development [22]. Having temporal views that allow us to understand how the community has evolved is quintessential.

For instance, even when two developers worked the equal amount of time on a certain module, if the one has worked over the period of two years and the other has been working during the last two months, the latter developer is likely to know better about the current version of the module. This kind of information is important to identify to whom ask about the module [24].

(2) Such data should support not only views for the summaries and overviews of the interaction history data but support ego-centric views, those based on individuals' perspectives.

Because it is situated within a social context, knowing the current state and its history of one's relationships with artifacts, the other developers, and the community, is not as straightforward as it seems. Such relationships are by no means objectively countable or measurable. One could only assume, or feel, what the relationships currently are and have been. One could also assume, thereby, how the relationships would look by another developer.

For instance, you think you have the X amount of expertise on a particular part of the source code. You may think that you are a little bit overestimated by one of your colleagues, Bob, and have a feeling that Bob thinks that you have the Y amount of expertise on the part. You think that Bob has the Z amount of expertise on the part, but again, Bob might think a little differently.

Thus, such technologies that support a community's metabolic process should help an individual to feel or assume the current state as well as its history of his/her relationships with artifacts, the other developers, and the community. They need to aim at providing data not only from an objective standpoint, but also from an individual, ego-centric viewpoint.

(3) Such data can be collected within the scope of a single community activity, as well as from that of external activities.

People's social relationships might be determined not only through activities within the community but also through those external to the community or within another community [43]. A software developer might be a member of another project, belonging to multiple communities.

A developer might be able to better understand the skill level of his/her peer by knowing the role of the peer member within another development community.

(4) Some parts of such data should only be partially disclosed to the community members, creating asymmetric information disclosure.

Software developers may not want to disclose all the historical information of his/her activities within the community. He/she should be able to explicitly specify some of the properties of his/her relationships with artifacts, developers and the community (e.g, the skill level with a certain module) because it is not always possible to adequately assume how such relationships are and have been.

The Saori system [15] provides users with awareness of and control over the information dissemination process within social networks. Saori allows users to specify types of information to be shared and a sharing policy at the level of mostly public and mostly private, not at the level of individuals. The STeP_IN (Socio-Technical Platform for in situ Networking) system [29] allows users to explicitly specify with whom developers want to communicate in what topics. This information is kept invisible to the other developers.

6. Social Factors

This section briefly examines social factors that affect software development driven by a knowledge community: motivation and interruption.

6.1 Motivation

Studies have recently been reported on how to motivate people to make contributions of higher quality to community-maintained artifacts of lasting value (CALVs). Ludford et al. [21] reports that telling people how they are special with respect to the group and its purpose increases member contributions and levels of satisfaction. Cosley et al. [4] argues that what they call "intelligent task routing," which is matching people with work, can be helpful to increase people's contribution, and that such intelligent task routing should consider not only the community's needs but also a person's knowledge and ability. Rashid et al. [33] has found that giving feedback about the value of a participant's contribution in terms of a small group the user has affinity with is most effective in motivating people to contribute.

Although the domain of these projects is movie recommendation and not software development, these findings seem to be equally applicable to software development as a collective creative knowledge community activity. On the other hand, this domain has fundamentally different nature from that of software development. In making a community repository of movie recommendations, the members of

the community has no clear purpose of *finishing it* having no explicit incentives for doing so. With many of software development projects, developers of each community share the clear goal of finishing a project, and they may be more motivated to help one another.

In either case, we need to conduct empirical studies to draw any significant conclusions on this matter, and further studies are necessary on how to motivate developers to contribute high quality artifacts and sustain the community as a system of evolution.

6.2 Interruption

Although interruptions between humans have mainly been studied in face-to-face communication settings, many findings seem to also be applicable to communications through email. In a face-to-face communication, an asker and a helper first need to go through a negotiate process making an agreement on when to interrupt the helper. People use a variety of social cues to decide when to start the negotiation process and making an agreement [42].

In using email for communication, it is much easier for a helper to ignore email message that asks for help. On the other hand, it is more difficult for an asker to get a timely help as the asker cannot tell when a helper would reply to the message. Studies by LaToza [20] found that this makes developers to go more and more face-to-face communications rather than using email, which causes serious problems of interruption especially employing agile development styles.

Wiberg and Whittaker [42] report that in their face-to-face interruption studies, users preferred to take interruptions as soon as possible. People preferred to take interruptions now, incurring the cost of disrupting their currently activity in order to avoid the future overhead of having to schedule and remember later commitments to talk. The authors also argue that users felt a social obligation to return calls and a need for being polite rather than delegating them even though it require more effort to do this.

These phenomena seem to also hold true for email communications. Although it is not as socially critical as in face-to-face communication, putting off replying to information-seeking messages often makes one to feel guilty. One may feel that he/she wants to reply to a message as soon as possible so that he/she would not need to worry about not forgetting replying.

To address this issue, the STeP_IN system [29][44] uses a mechanism to automatically set up anonymously addressed mailing list for an asker's request. The tool produces such a mailing list by taking into an account who is asking what question (i.e., the topic) and identifying a several set of developers in a community who have expertise in the topic and have *good* social relationships with the asker. The mechanism allows

receivers of the message to remain anonymous, letting them from not feeling bad by not replying to the message. When one of the recipients replies to the message, the identity of the helper is revealed to the asker and the regular ways of social interaction will follow, helping them to develop feelings of *expectation* and *obligation*. The approach is unique where the cost of interruption is treated as a collective manner. This aspect needs to be studied further in order to better support software development as collective creative knowledge work.

The field of human-computer interaction has long been studying how to model interruption between humans and computer agents [18][5]. Some parts of their models and findings should be taken into account to achieve more effective, less disturbing communication channels in support of software development within a social setting. For instance, one possible approach is to model the timing of when a potential helper should receive an email message by deliberately delaying the message delivery.

7. Related Work

The previous sections list existing tools and studies that address specific aspect of the approach. This section addresses three projects that have similar research goals with us in the domain of supporting software development as social activities.

The Augur system can be viewed as an example technique to look at software development as the system of evolution. The Augur system [6][13] simultaneously visualizes the structure of a software system (i.e., artifacts) and the structure of the development process carried out by developers (i.e. developers and the community). Augur visualizes the result of call graph analysis, and networks of contributors to a project, relating those who worked together on a single module. By looking at how developers worked together on what parts of a software system, a user of Augur could tell how relationships between artifacts (software system module structures) and developers change over time, including phenomena such as types of projects, how different roles different developers take, how such roles shifts between core and periphery, how authorship changes, and what patterns of stability and changes are observable. Augur currently supports ways to view the structural changes from an objective standpoint. Providing ego-centric individual viewpoints, for instance, from a particular developer's point of view, such as similar to the ones provided by SoyLent [9].

Another example is Hybrid Networks [23], which integrates links from multiple development data sources. The tool uses the Probabilistic Latent

Semantic Indexing clustering technique to associate and cluster data from email discussions, authors, and CVS source code tree branches. The result is integrated and displayed in a single visualized view. The tool currently does not support temporal views or ego-centric views.

As mentioned above, Storey et al. [36] argues for the importance of supporting awareness in software development by visualizing artifact and activity data, and report the result of comparing then-existing 13 tools that support such awareness. They have developed a survey framework, which consists of intention of the visualization, information that are visualized, presentation used in the visualization, interaction provided for the visualization, and effectiveness of the visualizations. Some parts of the framework, such as whether tools address temporal and historical changes over time, and what types of artifacts tools support, are important for our purpose. However, the framework does not focus on the relationships among artifacts, developers and the community, and how they change over time.

8. Discussion

Human aspects of software development have long been not highly focused [30][7] except in few approaches, such as empirical software engineering [2] and considerations of cognitive aspects of software engineering [16]. Recent trends in software engineering cannot be taken into a full account without seriously taking the social aspect of knowledge-intensive software development as a central theme. Using open source software, adapting agile methods through incremental change, and engaging in global software development equally aware of the importance of the collective, creative aspect. This would demand us to develop inter-disciplinary research agenda to cope with the issue. We as researchers and practitioners in this field need to engage in socio-technical collaboration for ourselves.

Acknowledgements

This research is partially supported by the Ministry of Education, Science, Sports and Culture (MEXT) Grant-in-Aid for Exploratory Research, 17650038, 2005.

Reference

[1] Adamczyk, P.D., Bailey, B.P., If not now, when?: the effects of interruption at different moments within task execution, Proc. . CHI04, ACM Press, pp.271-278, 2004.

[2] Basili, V., The Role of Experiments in Software Engineering: Past, Current, and Future, Proc. ICSE'96, pp.442-449, ACM, 1996.

[3] Coleman, J.S., Social capital in the creation of human capital. *American Journal of Sociology*, 94: pp. S95-S120, 1998.

[4] 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.

[5] Czerwinski, M., Horvitz, E., Wilhite, S. 2004. A diary study of task switching and interruptions, Proc. CHI'04, ACM Press, pp.175-182, 2004.

[6] de Souza, C., Froehlich, J., Dourish, P., Seeking the source: software source code as a social and technical artifact, Proc. GROUP05, ACM Press, New York, NY, pp. 197-206, 2005.

[7] Dittrich, Y., Doing Empirical Research on Software Development: Finding a Path Between Understanding, Intervention, and Method Development, *Software Practice is Social Practice, Social Thinking - Social Practice*, Dittrich, Y., Floyd, C., Klischewski, R. (Eds.), pp.243-262, MIT Press, 2002.

[8] 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.

[9] Fisher, D., Dourish, P., Social and temporal structures in everyday collaboration, Proc. CHI04, p.551-558, Vienna, Austria, 2004.

[10] Fisher, D. Understanding Communication Using Social Networks. *IEEE Internet Computing*. September/October, 2005.

[11] Fisher, D., Ask Not for Whom the Visualization is Rendered; It is Rendered for Thee. Workshop paper, presented at the Social Visualization Workshop, CHI 2006.

[12] Fisher, D., Smith, M., Welser, H. You Are Who You Talk To, Proc. HICSS, January 2006.

[13] Froehlich, J., Dourish, P. 2004. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. ICSE'04. IEEE Computer Society, 387-396.

[14] Gilbert, E., Karahalios, K., LifeSource: two CVS visualizations. CHI '06 Extended Abstracts on Human Factors in Computing Systems ACM Press, 791-796, 2006.

[15] Goecks, J., Mynatt, E. D. Leveraging social networks for information sharing. Proc. CSCW '04. ACM Press, 328-331, 2004.

[16] Herbsleb, J. D., Beyond computer science. Proc. ICSE '05. ACM Press, 23-27, 2005.

[17] Hook, K., Benyon, D., Munro, A.J. (Eds.), *Designing Information Spaces: The Social Navigation Approach*, CompSpringer, 2003.

- [18] Horvitz, E., Apacible, J. 2003. Learning and reasoning about interruption. Proc. ICMI '03. ACM Press, pp.20-27, 2003.
- [19] Iqbal, S. T., Bailey, B. P., Leveraging characteristics of task structure to predict the cost of interruption, CHI'06, ACM Press, 741-750, 2006.
- [20] LaToza, T.D., Venolia, G., DeLine, R. Maintaining mental models: a study of developer work habits, Proceeding of the ICSE '06. ACM Press, 492-501, 2006.
- [21] Ludford, P.J., Cosley, D., Frankowski, D., Terveen, L., Think different: increasing online community participation using uniqueness and group dissimilarity, Proc. CHI'04, ACM Press, 631-638, 2004.
- [22] Maturana, H.R., Varela, F.J., The Tree of Knowledge: The Biological Roots of Human Understanding, Shambhala Publications, Inc., Boston, MA, 1998.
- [23] Medynskiy, Y., Ducheneaut, N., Farahat, A., Using hybrid networks for the analysis of online software development communities, Proc. CHI'06, ACM Press, 513-516, 2006.
- [24] Mockus, A., Herbsleb, J. D., Expertise browser: a quantitative approach to identifying expertise, Proceedings ICSE'02. ACM Press, 503-512, 2002.
- [25] Nahapiet, J., Ghoshal, S., Social Capital, Intellectual Capital, and the Organizational Advantage. Academy of Management Review, 23, pp.242-266, 1998.
- [26] 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.
- [27] 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.
- [28] Nakakoji, K., Humane Requirements for Enabling and Nurturing Collective Creativity, Proc. HCI05, Las Vegas, NV, CD-ROM, Jul. 22-27, 2005.
- [29] Nishinaka, Y., Asada, M., Yamamoto, Y., Ye, Y., Please STeP_IN: A Socio-Technical Platform for in situ Networking, Proc. APSEC'05, Taipei, pp. 813-820, Dec. 2005.
- [30] Noerbjerg, J., Kraft, P., Software Practice is Social Practice, Social Thinking - Social Practice, Dittrich, Y., Floyd, C., Klischewski, R. (Eds.), pp.205-222, MIT Press, 2002.
- [31] Resnick, P. Beyond bowling together: sociotechnical capital. Carroll, J. M. (Ed.) HCI in the New Millennium, pp. 247-272, 2002.
- [32] Rajlich, V., Changing the Paradigm of Software Engineering, Communications of ACM, Vol.49, No.8, pp.67-70, August, 2006.
- [33] Rashid, A. M., Ling, K., Tassone, R. D., Resnick, P., Kraut, R., Riedl, J., Motivating participation by displaying the value of contribution, Proc. CHI'06, ACM Press, New York, NY, 955-958, 2006.
- [34] Robillard, P. N., The role of knowledge in software development. Comm. ACM 42, 1 87-92, 1999.
- [35] Sillito, J., Murphy, G., De Volder, K., Questions Programmers Ask During Software Evolution Tasks, Proc. Symposium on Foundations of Software Engineering, November 2006 (to appear).
- [36] Storey, M-A. D. Cubranic, D., German, D.M., On the use of Visualization to Support Awareness of Human Activities in Software Development: a Survey and a Framework, Proc. SoftVis'05, ACM Press, pp.193-202, 2005.
- [37] Szoestek, A.M., Markopoulos, P. Factors Defining Face-To-Face Interruptions in the Office Environment, CHI2006, Work-in-Progress, pp.1379-1384, 2006.
- [38] Terveen, L., McDonald, D. W. 2005. Social matching: A framework and research agenda, ACM Trans. of Comput.-Hum. Interact. 12, 3, 401-434, 2005.
- [39] Viegas, F., Smith, M., Newsgroup Crowds and Authorlines: Visualizing the Activity of Individuals in Conversational Cyberspaces, HICSS-37, Hawaii, January 2004.
- [40] Wenger, E., Communities of Practice - Learning, Meaning, and Identity. Cambridge, England: Cambridge University Press, 1998.
- [41] Westrup, C., On Retrieving Skilled Practices: The Contribution of Ethnography to Software Development, Social Thinking - Social Practice, Dittrich, Y., Floyd, C., Klischewski, R. (Eds.), pp.95-110, MIT Press, 2002.
- [42] Wiberg, M., Whittaker, S., Managing availability: Supporting lightweight negotiations to handle interruptions. ACM Trans. of Comput.-Hum. Interact., 12,4, 356-387, 2005.
- [43] Ye, Y., Yamamoto, Y., Dynamic Communities in Support of Situated Knowledge Collaboration, Proceedings HCI05, Las Vegas, NV, CD-ROM, Jul. 22-27, 2005a.
- [44] Ye, Y., Dimensions and Forms of Knowledge Collaboration in Software Development, Proceedings APSEC, Taipei, pp. 805-812, Dec. 2005b.

When Programmers Don't Ask

Sukanya Ratanotayanon
Department of Informatics
University of California, Irvine
sratanot@uci.edu

Susan Elliott Sim
Department of Informatics
University of California, Irvine
ses@ics.uci.edu

Abstract

Throughout the software development process, participants of the project need to collaborate in order to exchange the knowledge required to complete the project. Exchanging and obtaining knowledge is often done through asking and answering questions. We present an initial study aimed at understanding question-asking behavior during knowledge exchange in software development. We found that this seemingly simple activity is often not performed well, nor as frequently as required. Novices do particularly poorly as they are not aware of their knowledge needs. Experts also asked few questions but focused on different kinds of knowledge. In addition, they sometimes, ask questions although they have ability to obtain information themselves. We speculate on the causes of failures in question asking and the rationale behind experts' questions.

1. Introduction

Software development is a knowledge-intensive activity. Various stakeholders who are involved in the project need to collaborate and communicate in order to exchange and transfer their knowledge. However, collaboration and knowledge exchange in software projects are often not performed effectively. As reported by Curtis [1], the most common and severe issues in software development projects are the thin spread of domain knowledge, and communication and coordination breakdowns. These issues need to be addressed in order to improve overall development process performance.

A common way to exchange knowledge is through asking and answering questions. However, in order to effectively perform this seemingly simple activity, the participants need to be aware of what knowledge others have and their needs for knowledge. In addition, as observed in other fields such as education, it is not

unusual that a person does not ask questions because he lacks the information or does not understand the given explanation. If this behavior is also present in software development, it could decrease the effectiveness of knowledge exchange.

In order to support knowledge collaboration, we need to understand the question-asking behavior: i) how and why people ask questions and ii) what kind of information is needed, including its importance to the inquirer's knowledge needs. The answer to these questions can provide guidance for designing collaboration tools, such as what kinds of information is needed but is not requested, or support for novices to ask the right questions.

In this paper, we report on an initial exploratory study in a laboratory setting. The data reported here comes from a multi-part study in which novice and expert software engineers are required to collaborate on a change request for a moderate-sized web application. Our goal is to explore the question-asking behavior in knowledge transfer. We observed how software developers ask questions in a collaboration session to increase their understanding of the application and to complete an assigned task.

We found breakdowns in question asking. Our results showed that developers do not ask questions well, nor do they ask as often as they should. Novices did especially poorly as were not aware of their knowledge needs and didn't ask questions when they really should have. Experts also asked few questions, but focused on knowledge from different levels of abstraction. In addition, they sometimes, asked questions that they have ability to answer themselves.

This paper is organized as follows. Results from related research are reviewed in Section 2. The laboratory procedure used in the study is described in Section 3. The results are presented in Section 4. The discussion of the causes of the failures in question asking is presented in Section 5. Section 6 presents our future work. We conclude our paper in Section 7.

2. Related Work

Previous studies in question-asking fall into two main categories: psychological studies and field studies of question-asking in software development and engineering.

In psychology, questions tend to be used as an indicator of cognitive activity. Questions are viewed as a means to obtain knowledge that is important for the inquirer to reach a certain goal [2, 3]. Ram suggested that questions are crucial and central to learning [3]. The question formulation process is the process of identifying what the learner needs to learn, and asking the right questions allows the learner to focus on relevant issues by pursuing the questions. Therefore, the depth of understanding of the learner depends on the questions asked. However, as shown in experiment performed by Miyake and Norman [4] a prerequisite for asking questions about a new topic is an appropriate level of knowledge with which to formulate the question and to interpret the response. The number of questions a person asked when learning new material depended on two variables: i) the existence of a proper knowledge structure and ii) the level of completeness of those structures regarding new material.

Although, the work discussed above gives us an insight into question asking as a cognitive activity, it doesn't address what questions are asked and how they are actually asked in a work situation. The following are field studies investigating questions asked in software development and engineering.

Berlin performed field study of consulting interaction between apprentices and experts [5]. The results showed that question-asking played an important role in collaborative conversation between apprentices and experts. Confirmative questions were used by experts to invite apprentices' interjections. Apprentices used questions that restated the explanation to signal their level of understanding and to ask for validation or help. This collaborative process was important to providing a successful explanation, because the pair continually sought and provided evidence that they understood each other, which resulted in rapid repairs of misunderstandings. Berlin also found that experts were quicker to seek help from other experts, which might be due to better self-monitoring skill or social factors, such as having more reciprocal relationship with other experts.

Herbsleb et al. performed a field study aimed to assess knowledge needs in software development by examining the questions asked in requirement specification and design meetings [6]. Data was collected from projects in requirements and early

design phases. The results showed that the most common questions were "what" and "how" questions targeting requirements, even for the project that already move into design phase. Very few "why" questions were asked although design rationale [7] is considered very important information.

Ahmed and Wallace studied queries made by novice and experienced designers in a large aerospace company. Similar to Herbsleb, the goal of the study was to identify knowledge needs of designers and their awareness of their knowledge needs. The results showed differences between novices and experts in both types of queries made and patterns of responses to the queries. In addition, the finding indicated that novice designers tended to be unaware of their knowledge needs and required support in identifying what they needed to know.

3. Empirical Method

The goal of our study was to gain an understanding of question-asking behavior as a means of knowledge transfer in software development projects. We observed how software developers asked questions to increase their understanding of the application in order to complete an assigned task.

3.1. Research Design

While quantitative studies use experimental methods and quantitative measures to make predictions and generalize findings, qualitative studies use a naturalistic approach to build understanding and extrapolate to similar situations [8]. In order to gain a better understanding of question-asking behavior, we chose to perform an exploratory qualitative study instead of a quantitative study which ignores effects that may be important, but are not statistically significant. Qualitative methods provide a wider understanding of the entire situation as it accepts the complex and dynamic quality of the social world. It provides results that are more in-depth and comprehensive than those produced by quantitative methods.

We performed a laboratory study simulating a situation in software maintenance. An existing developer has to transfer his knowledge to assist another programmer who is also assigned to make a modification to a software system. This allows us to evaluate the quality and relevance of questions asked by subjects. More detailed information of study design is presented elsewhere [9].

3.2. Procedure

Two subjects were required to participate in each session. Each session took a total of 210 minutes and comprised three tasks: Task A, Handover and Task B. The time line of each session is depicted in Figure 1.

	0:30	1:00	1:30	2:00	2:30	3:00	3:30
Subject A	Task A			Handover	Task B		
Subject B	Handover			Task B			

Figure 1: Time line of study procedure

Task A: The first subject was given a scenario where a customer requested a feature in the company’s survey management application. He was asked to complete a Change Request Proposal (CRP) form describing how to make the change. The CRP gives the guideline of what information should be provided to Subject B. The subject was also asked to not make any modification and had up to 90 minutes to finish this task individually.

Handover: In this task, Subject A verbally handed over to Subject B the information gathered in the first task. The Handover task began with Subject A giving an explanation without any interruptions. During the explanation, the application and its code might be shown to Subject B to improve his understanding. Once the explanation was completed, Subject B was allowed to ask questions. The subjects were given 30 minutes to perform this task.

Task B: Following the Handover, Subject B was left with the CRP and Subject A’s notes. Subject B had to work individually and make the modification within 90 minutes.

This division of tasks not only allows us to examine the collaboration behavior of asking question, it also mimics common work situations where research is separated from detailed work. The CRP is a commonly used process in which software evolution is managed by a Change Control Board (CCB).

Before performing the tasks, both Subjects were given a short description of typical architecture of web applications, task description and instructions on running and compiling the application. Subjects were allowed to use any information available on the Internet. There was no application developer documentation, such as a design document. However, this omission is not uncommon in real-world maintenance settings.

During each session, each subject’s activities were recorded by a web camera, a microphone and screen capturing software. Scratch paper was provided to the participants and was collected at the end of the study.

Eclipse IDE containing a project set up for the task and TextPad software were also provided.

3.3. Software characteristics

The application used in this study was an open-source web-based survey management tool called VTSurvey, developed at Virginia Tech. It is a typical n-tier web application created with JSP, Java, and XML technologies, and runs on the Tomcat application server. It enables users to create, maintain and run online surveys. It also provides a user management system for managing user accounts. Originally, VTSurvey didn’t maintain each user’s email address. We requested that the system be modified so that it can save and display user’s email addresses.

VTSurvey consists of 38 Java™ files, 74 JSP (Java Server Page) files. In addition, there were 4 DTD (Document Type Definition) files. It has a total of 10,342 lines of code. Subjects were presented with all source files, including those that were not relevant to the assigned task.

3.4. Subjects

The subjects were mainly recruited by word of mouth. A total of twelve subjects participated in the study. Half the subjects were novices and the other half experts. Novices were senior undergraduates, or recent graduates who had been working for less than one year. Experts were developers with five or more years of work experience. We had three female subjects and all of them were experts. All subjects considered themselves fluent in English and had experience working with Java. We also surveyed their experience (including non-work experience) in the areas of web development and database management. Figure 2 presents overall level of experience of subjects in each group.

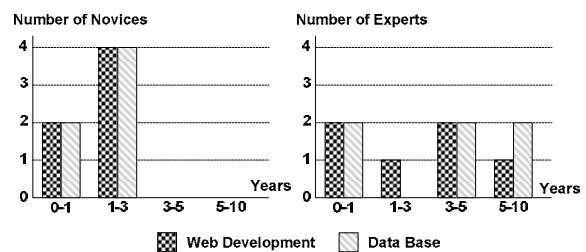


Figure 2: Level of experience in Web Dev. and DB

4. Analysis and Results

We analyzed audio and video data recorded during the six sessions. The implementation was graded based on its correctness and completeness regarding our requirements and was given a numeric score out of 55. The conversations during Handover sessions were transcribed in order to identify questions from both subjects. We included both implicit and explicit questions in our analysis. Explicit questions were formed in question sentences. The implicit questions were formed as normal sentences or fragments, but acted as questions due to the inquirer's expression and tone, and generated answers from the partner. The full list of questions is found in Table 3 and Table 5.

4.1. Number of Questions

We identified only total of 64 questions from the six sessions. Due to time constraints on the implementation task, we expected that Subject B would ask a lot of questions in order to take advantage of Subject A's knowledge. However, there were surprisingly few questions asked, in comparison with other studies [10, 11]. More than half of these questions were asked in a single session, the fourth one. In addition, the purpose of the questions, as well as the quality and type were very different from other sessions. As a result, we decided to analyze this session separately and this is presented in Section 4.4. The number of questions asked and total time spent in the Handover sessions are shown in Table 1.

Table 1: Number of questions and time spent

Run	Subject A	Subject B	Total	Handover Time (min)
1 (E-N)		5	5	9:55
2 (E-N)	1		1	3:00
3 (N-E)		7	7	7:44
5 (N-N)		5	5	10:00
6 (E-E)	4	8	12	12:45

4 (N-E)	10	24	34	27:56
---------	----	----	----	-------

* (Subject A - Subject B), E = Expert, N = Novice

4.2. Types of Questions

In order to perform the modification, the subjects needed to understand the VT survey application. Previous studies [12, 13] showed that in order to successfully comprehend software, information from different levels of abstraction is needed in order to build a mental model. To investigate what information subjects asked for, we categorized the knowledge requirement of the questions into three levels of

abstraction. The summary of the categories is shown in Table 2.

Domain Level (**D**) questions are concerned about a program's external behavior, such as those perceived by users of the application. We further categorized questions in this level into:

D1: Questions that ask about task's requirements, as provided in the task description document in order to clarify or to confirm them.

D2: These questions ask about what the application can do and how a user performs a specific task.

D3: These questions aim to confirm the scope of the task. The inquirers express concerns whether additional features are required to complete the assigned task.

Intermediate Level (**I**) questions ask about mechanisms that map between domain and program level behaviors. Examples of this kind of information includes: software architecture and high-level design information.

Program Level (**P**) includes questions about low-level design or information that is often grounded in the source code. These questions are divided into:

P1: Questions about location of data files, source code files and how to run the application.

P2: Question about meaning and behavior of methods, objects, or JSP files. These questions target the information in a lower level of abstraction than question in **I** category.

All other questions such as those asking about tool preferences are grouped in the Other (**O**) category.

Table 2: Summary of categories of questions

Level	Category
Domain	D1: Restatement of Change Request. D2: Application Usage D3: Task Scoping
Intermediate	I: Mechanics
Program	P1: Set Up P2: Code Level Mechanics
Other	O: Questions that were not asked about the application.

All questions asked in each category except those from the Run 4 are presented in Table 3 grouped by the run in which they were asked.

Table 3: All questions asked excluding run 4's

Questions	
D1: Restatement	
R1	<ul style="list-style-type: none"> ▪ So all they wanted was? ▪ What (field) you want to see is? ▪ (you need to display) Email address. Not the password?

R3	<ul style="list-style-type: none"> Is the user id basically any characters or numbers? Save (email) the same way (as user id and password)?
R6	<ul style="list-style-type: none"> Just add. Right? Add this field.
D2: Usage	
R1	<ul style="list-style-type: none"> They (users) can't edit?
R6	<ul style="list-style-type: none"> So this one (UI) is add new user and this one (UI) is change?
D3: Scoping	
R3	<ul style="list-style-type: none"> Do I have to verify that it (email field) is blank? No other fields (to be added)? Do I have to require the person to answer in email twice?
R5	<ul style="list-style-type: none"> I have to check for email and make sure it was entered into form?
R6	<ul style="list-style-type: none"> I'm not sure if we want to include this piece of information?* Should we allow the admin to change the user's email or not?* I'm not sure whether they require us to do this or not?* Whether the email information is require or not ... [if the blank data is allowed]?* [Do I have to validate incorrectly formed email address]?
I: Mechanics	
R3	<ul style="list-style-type: none"> And surveyMetaData is like the database structure?
R6	<ul style="list-style-type: none"> So they don't need... they don't use data base right?
P2: Code-Level Mechanics	
R1	<ul style="list-style-type: none"> On the backend there is no password field?
R3	<ul style="list-style-type: none"> What are these files?
R5	<ul style="list-style-type: none"> Which one talks to the user.dtd? So it is just those three things I'm going to worry about? Which files are going to need to be modified?
R6	<ul style="list-style-type: none"> Is this file going to control the elements in the xml file? I'm not sure whether we can use this new data structure to process the old data?* Leave what? There's going to be a pair in the xml files right? So what files are involved in these changes?
O: Other	
R2	<ul style="list-style-type: none"> Whether you are familiar with the code?*
R5	<ul style="list-style-type: none"> Should I just save it (currently opened files)?

* shows questions asked by Subject A

The majority of the questions were at the Domain level, although our requirements were quite simple. In addition, there were no "why" questions asked although design rationale is considered important information in comprehending software. This result is similar to the finding from Herbsleb et al. [6].

To completely understand a program, a programmer has to establish links between information in different levels of abstraction. Incomplete explanations and few questions suggested that a lot of information may be missing. To our surprise, there were few questions

asked at the Intermediate and Program level regardless of the completeness of the explanation received. Further examination of the implementations showed that subjects had difficulty finding out the information by themselves, especially for the novices. This suggests that subjects did not already possess the information that was not asked for. In addition, the coverage of question was very narrow (See Section 4.4 for comparison).

We also expected that Subject A would ask about his partner's experience in order to provide explanation in a suitable presentation level. However, there was only one question of this kind.

Table 4: Number of questions in each category and implementation score of each session

Run/ Level	1 EN	2 EN	3 NE	5 NN	6 EE	Total	4 NE
D1	3		2		1	6	
D2	1				1	2	1
D3			3	1	5	9	
I			1		1	2	13
P1						0	5
P2	1		1	3	4	9	5
O		1		1		2	10
Total	5	1	7	5	12	30	34
Impl.	35	12	35	50	44		55

The number of questions asked in each category and the implementation score from each session are presented in Table 4. In this study, subjects who asked about information in the I and P levels were better able to complete the task than those who did not. This is reasonable as our task had a time limit. Asking for this information allowed the subjects to exploit knowledge from his partner and to spend less time doing research. The fact that they were able to formulate the questions suggested that they would be able to understand the explanation and hence able to make use of the information.

Over the next two subsections, we discuss two interesting patterns of questions-asking that were manifested in our study.

4.3. Run 4: Asking in the Absence of Answers

More than half of questions in the entire study were asked in Run 4 by our most experienced subject, E4, who had 5-10 years of experiences in both web applications and databases. In addition, E4 was the only subject who completed the modification task. E4 received a very short and incomplete explanation from his partner, N4, who had less than one year of

experience in both areas. The complete list of questions asked in Run 4 is presented in Table 5.

A large number of questions might seem reasonable as E4 had sufficient knowledge to detect missing information and could ask questions to obtain the missing information. However, what made this session interesting is that E4 decided to continue “asking” N4, who was obviously a novice and could not answer even his simplest questions. The following excerpt was taken from the beginning of Run 4’s Handover session.

E4: *Where is the data store?*

N4: *No idea. I don’t know where the data is stored. I can’t even find servlets.*

E4: *Hmm. Why couldn’t you find the servlets?*

N4: *I don’t know where it’s at.*

E4: *Why don’t you know?*

It would not have been surprising if E4 disregarded N4’s explanation and ended the Handover session at this point. But E4 persisted. This question-asking pattern continued for nearly half an hour. When N4 could not answer the questions, E4 found the answers to his questions and explained them to N4. Since E4 clearly could not obtain answers from N4, what was the purpose of questions in this session?

As some information provided by N4 was incorrect, E4 used questions to judge the correctness of explanation provided by N4. In addition, questions were used to engage N4 in the collaborative conversation. For example, after examining a portion of Java code E4 posed a question:

E4: *Doesn’t it gives you the impression that each files has a user associated wit it?*

N4: *Name of the user as a file?*

E4: *The user yeah, the name of the user is the name of the file.*

N4: *As you pointed out, it’s an array of*

In addition, the type and quality of questions asked in this session were different from the other sessions. There was only one Domain level question and most of questions asked were in Intermediate level. In addition, the questions touched on more parts of the system in more detail.

Table 5: Questions asked in Run 4

Run	Questions
	D2: Usage
R4	▪ Where to actually go to that... the main page?
	I: Mechanics
R4	▪ Did you say I do or do not have to modify the Servlet? ▪ It is a Servlet. It’s not a javabean, not ... ?

	<ul style="list-style-type: none"> ▪ Where is the data stored? ▪ Why couldn’t you find the servlets? ▪ Why don’t you know (where the Servlet is)? ▪ Do you know if the data is stored in an xml file or a real relational database? ▪ Doesn’t it give you the impression that each file has a user associated with it? ▪ The name of the user as a file?* ▪ What give you the impression that there are Servlets involved and not just jsp files? ▪ Did you see something like that (how Servlet is used by Subject A in his past experience) here? ▪ So how would that (using only JSP to implement web application) work anyway?* ▪ How would what work? ▪ Do you know if it’s using the standalone or LDAP authentication method?
	P1: Set Up
R4	<ul style="list-style-type: none"> ▪ Do you know where that directory is? ▪ Do you know the name of one of the user on the system? ▪ Do you know where the user class is? ▪ Do you know where the source file is? ▪ Do you know how to build this stuff? Have you done that?
	P2: Code-Level Mechanics
R4	<ul style="list-style-type: none"> ▪ I don’t know what that (HttpUtils.getRequestURL(request)) is ?* ▪ How does the listAllUser.jsp file retrieve the email address or retrieve the user name for display? ▪ Where would you go? I mean the action... * ▪ It returns back to the page, which make sense. Right? ▪ If you get down to “setPassword”, what does it do?
	O: Other
R4	<ul style="list-style-type: none"> ▪ Have you seen the program?* ▪ You understand this part. I mean why you have to change it?* ▪ Can you do find for “test” (a user name)? ▪ Can you add another user in the system with a more unique name? ▪ Can you open the xml file? ▪ This (file that he was asked to open) one?* ▪ What do you want to see it (the file) with?* ▪ Could we look at the constructor?* ▪ How long will it take you to do that?* ▪ Localhost... Can we go back to the main ... main page?

* shows questions asked by Subject A

4.4. Run 1 and 2: Novices Don’t Ask Experts

In contrast to Run 4, very few questions were asked in Runs 1 and 2. These sessions had expert explainers and novice implementers, and very few questions were asked especially in Intermediate and Program level. However, the quality of explanations given in both sessions was different.

In Run 1, N1 received a very good explanation from E1. N1 asked very few questions, mostly in the D1 category. At first glance, this suggests that N1 received all the important information needed to complete the task down to the level of which files and methods were needed to be modified and how to modify them. There was no need to ask questions to obtain further information. However, this was not the case. The video from implementation task showed that N1 did not understand the explanation given and had difficulty utilizing the information given by E1. N1 disregarded the CRP from E1 and spent a lot of time trying to obtain the same information available in that document.

In Run 2, E2 gave a very short and incomplete explanation to N2. However, N2 did not ask *any* questions. From his implementation task result, we know N2 had trouble finding information on his own. In addition, E2's explanation contained some incorrect information; N2 ended up being misled and received the poorest implementation score.

It is obvious that both novices did not have the information required to understand the application. Why didn't they ask more questions to their partners who are experts and should be able to provide them with useful information? In the next section, we explore some possible explanations.

5. Discussion: Why developers don't ask

In this study, the Handover sessions were rather short and there were surprisingly few questions asked, especially by novices. We expected Subject B to ask for more information in order to benefit from Subject A's knowledge. In this section, we will speculate on the reasons why there were so few questions.

5.1. Experts

The result showed that experts were responsible for asking the bulk of questions in this study. Excluding Run 4, 20 out of the 30 questions were asked by experts. This makes an average of five questions asked per person which is still not many. A possible explanation is that experts are confident that they can find out required information on their own. Among the 20 questions asked by experts, 8 were asked to novices and 12 to experts. This shows a tendency to ask more questions to experts than to novices. This might be because experts have more common ground with other experts. Also, experts can easily identify whether their partners are novices or experts [14]. When paired with novices, it's possible that experts could detect their

lack of experience and discounted their ability to provide useful answers.

The types of questions asked by experts were also different from novices. Experts asked more questions at the Program level than novices. This might be because experts have better domain models which allow them to know what information they need at P level. P level information will be used to work out what information is needed at the I level. This was also substantiated by Run 4's collaborative problem solving and information seeking.

5.2 Novices

In this study, excluding Run 4, novices asked average of only two questions per person. A possible explanation is that novices lack domain knowledge and a suitable framework to understand the explanation, an inability to know what they need to know [4]. This explanation is supported by Ahmed and Wallace's finding that novices usually don't understand their knowledge needs [11]. It is also possible that novices may have questions but did not know how to ask them, because they have difficulty framing question due to the lack of common ground with the explainer. As presented by Ram [3], in order to ask a question, one almost needs to know what the answer is going to be. This is similar to forming a hypothesis about the answer. Also, in order to understand an answer, one needs to be able to anticipate the answer in order to incorporate it into existing knowledge.

5.3 The Social Act of Asking Questions

People don't always ask their questions. The simple explanation is that they are afraid of asking stupid questions or they are self-conscious. However, the reasons for this are more varied and more profound.

Flammer proposed that the process of asking is a decision process that negotiates between costs and benefit of asking the question [2]. Examples of cost include time and effort spent in asking and understanding the answer, and the shame of appearing ignorant. Other factors include importance of questions, likelihood of existing answers elsewhere, and likelihood of understanding an answer. Our laboratory study de-contextualizes the interaction between subjects. They had no prior experience with the application and were not familiar with each other, which makes background and credibility of the information source unclear. Our subjects may have felt hesitant to ask questions because benefit of asking was unknown.

Cultural factors also affect questions-asking behavior. Berlin [5] observed that among developers, semantic questions were preferred over questions about "simple" technical problems with the environment, tools, or programming syntax. In addition, the culture also encouraged novices to try to find out answer on their own before asking experts. Asking "trivial" questions might be considered bothersome or distracting to experts, whose time is valuable. Finally, face-to-face question-asking may not be valued as a means to transfer and manage knowledge because it doesn't leave a record. The use of Instant Messaging tools or email might be a preferable way to ask questions.

6. Future Work

We plan to address open issues and to further observe the question-asking behavior in future studies. Possible modifications to the study design include: i) controlling the quality of presentation and explanation given to Subject B using confederates; ii) allowing Subject B to ask questions during the implementation; iii) providing additional means for asking follow-up questions through Instant messaging or email; and iv) use the debriefing session to ask about the reasons that a subject asked or didn't ask questions, and what they think they should or should not have asked.

7. Conclusion

We have presented an initial study aimed at understanding question-asking behavior in knowledge exchange in software development. The study required novice and expert software engineers to collaborate on a change request for a web application. We found breakdowns in question-asking by both novices and experts. The results showed that this seemingly simple activity is often not performed well nor as frequently as necessary for successful knowledge collaboration. Novices may not realize their knowledge needs nor be able to frame questions due to lack of domain knowledge. Other factors that prevent developers from asking questions are lack of common ground, likelihood of finding an answer without asking, a disbelief in the credibility of information source, and the low cultural value of a some types of questions.

8. Acknowledgements

Our thanks to Oluwatosin Aiyelokun, Erin Morris, Justin Beltran, Matt McMahan, Teerawat Meevasin, John Situ, Derrick Tseng, Jonathan Zargarian for

providing invaluable assistance in preparing and conducting the experiment. Thanks also to Jeff Elliott and Yuzo Kanomata for helping us with initial pilot testing.

This material is based upon work supported by the National Science Foundation under Grant No. 0430026. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

9. References

- [1] B. Curtis, H. Krasner and N. Iscoe, "A field study of the software design process for large systems," *Communication of ACM.*, vol. 31, pp. 1268-1287, 1988.
- [2] A. Flammer, "Towards a theory of question asking," *Psychological Research*, vol. 43, pp. 407, 1981.
- [3] A. Ram, "A Theory of Questions and Question Asking," *The Journal of the Learning Sciences*, vol. 1, pp. 273, 1991.
- [4] N. Miyake, D.A. Norman, "To Ask a Question, One Must Know Enough to Know What Is Not Known." *Journal of Memory and Language*, vol. 18, pp. 357, 1979.
- [5] L.M. Berlin, "Beyond Program Understanding: A Look at Programming Expertise in Industry," in *Empirical Studies of Programmers: Fifth Workshop*, pp. 6-25, 1993.
- [6] J.D. Herbsleb, H. Klein, G.M. Olson, H. Brunner, J.S. Olson and J. Harding, "Object-oriented analysis and design in software project teams," *Human Computer Interaction*, vol. 10, pp. 249-292, 1995.
- [7] A.L. Jarczyk P. and I.F. Shipman, "Design Rationale for Software Engineering: A Survey," *Proceedings of the 25th Annual IEEE Computer Society Hawaii Conference on System Sciences*, pp. 577-586, 1992.
- [8] M.C. Hoepfl, "Choosing qualitative research: A primer for technology education researchers," *Journal of Technology Education*, vol. 9, pp. 47, 1997.
- [9] S.E. Sim, S. Ratanotayanon, O. Aiyelokun and E. Morris, "An Initial Study to Develop an Empirical Test for Software Engineering Expertise," *UCI-ISR-06-6*, 2006.
- [10] M.M. Sebrechts and M.L. Swartz, "Question asking as a tool for novice computer skill acquisition," in *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 293-299, 1991.
- [11] S. Ahmed and K.M. Wallace, "Understanding the Knowledge Needs of Novice Designers in the Aerospace Industry," *Des Stud*, vol. 25, pp. 155-173, 2004.
- [12] A. Von Mayrhauser, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, pp. 44, 1995.
- [13] N. Pennington, "Comprehension strategies in programming," pp. 100-113, 1987.
- [14] R.L. Campbell, N.R. Brown and L. DiBello, "The Programmer's Burden: Developing Expertise in Programming," in *The Psychology of Expertise: Cognitive Research and Empirical AI*, R.R. Hoffman, New York: Springer-Verlag, 1992, pp. 269-294.

Recommending Library Methods: An Evaluation of Bayesian Network Classifiers

Frank McCarey, Mel Ó Cinnéide and Nicholas Kushmerick
School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland.
{frank.mccarey, mel.ocinneide, nick}@ucd.ie

Abstract

Programming tasks are often mirrored inside an organisation, across a community or within a specific domain. We propose that final source codes can be mined, that knowledge and insight can be automatically obtained and that this knowledge can be reused for the benefit of future developments. We focus on reusable software libraries; we wish to learn information about how such libraries are used and then elegantly pass this information onto individual developers.

In this paper we investigate a Collaborative Filtering approach of recommending library methods to a individual developer for a particular task. The central idea is that we find source codes that are the most relevant to the task at hand and use these to suggest useful library methods to a developer. To determine the similarity and relevance of source code, we investigate and compare a number of Bayesian clustering techniques including Bayesian Networks and Naïve-Bayes. We present results and discuss the suitability of Bayesian networks to this domain.

1. Introduction

A healthy knowledge flow between programming peers can positively impact personnel morale, team productivity and the ultimate outcome of a project. Be it a new developer just added to the team or the experienced professional unfamiliar with a particular library, all can benefit from the experiences and skills of others. The tools and techniques used to share such information within organisations can vary greatly; for example, colleagues may hold informal meetings, telephone or email each other or perhaps rely on detailed support materials. Though these techniques may be effective, it is clear that they lack efficiency. For both the requestor and the responder, there is the overhead of task switching; just replying to an email may upset the

flow of ones primary task. Similar to Ye and Fischer [28], we propose that much of this knowledge can be shared automatically through the provision of proper tool support.

In this paper we focus on tool support for software libraries. Reuse of such libraries has been shown to improve software quality and developer productivity whilst reducing defect density [20] and time-to-market [29]. It is impractical though to consider that any one individual would be entirely familiar with any one library; for example, the latest version of the Java API library has over 3000 classes while the Java Swing library has over 500 classes. Over a period of time, it is likely that many different programmers will have used a particular library. We suggest that insight can be gained from analysing how particular libraries are used and that this knowledge can be passed onto individual programmers through intelligent support tools; we are currently developing the RASCAL tool.

RASCAL is a proactive recommender that is designed to support library reuse. RASCAL hopes to address several of the pragmatic issues that currently hamper reuse; for example, developer motivation, time constraints, library accessibility and lack of conversancy for a particular library. RASCAL currently recommends a set of library methods to a developer which it believes to be relevant to the task at hand. We propose that by identifying and recommending reusable methods from a library and subsequently facilitating quick access to these, we will foster and encourage reuse.

Similar to many commercial recommenders, RASCAL produces a set of personalised recommendations for an individual. However, unlike other domains where perhaps a set of books or movies may be presented to a customer, RASCAL recommends a set of task relevant methods to a particular developer. Like most recommendation tasks, RASCAL recommends software methods that the developer is interested in. Recommendation in our tool is complicated though because we wish to recommend methods which we believe the developer may be unfamiliar with or unaware of. Another interesting distinction between our recommender

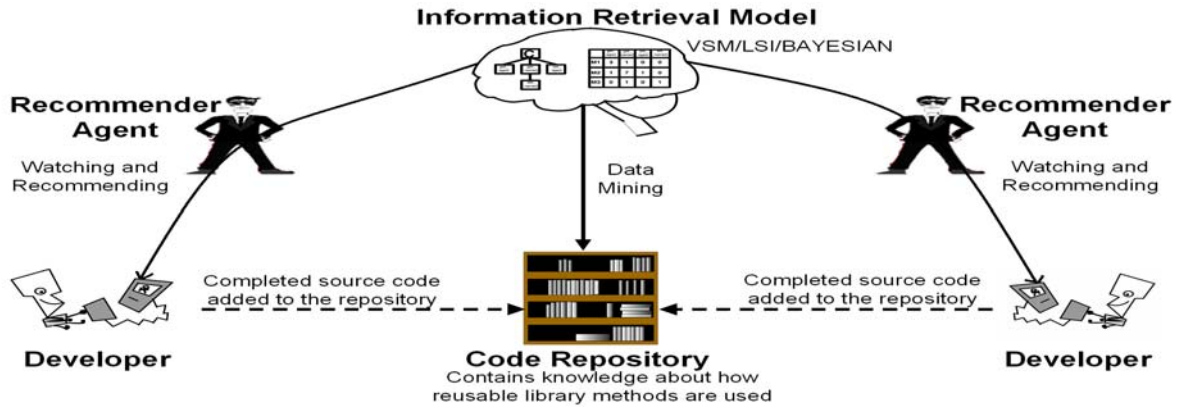


Figure 1: RASCAL Overview

system and most mainstream recommenders is that we are trying to predict, in order, the next likely method a developer will employ. Many typical recommender systems only predict a vote for items which the user has not yet tried. Our aim is to predict the next library method a developer should invoke; it is quite likely that the developer will have invoked this method previously.

Recommendations are produced using a Collaborative Filtering (CF) [25] algorithm as explained in section 3. An important aspect of CF algorithms is clustering users; in this paper we investigate and compare a number of Bayesian approaches that can be used to classify how similar source codes are. In particular, we will detail Bayesian Networks, Naïve-Bayes, Tree-Augmented Naïve Bayes, Forest-Augmented Naïve Bayes and finally Bayesian Network Augmented Naïve-Bayes.

The main contributions of this work are:

- A viable solution to domain knowledge sharing, in respect of software reuse libraries.
- A technique embedded in the RASCAL support tool that significantly enhances reuse.
- An investigation of how effectively Bayesian techniques can be applied to source code. We use these techniques to support reuse but in theory several other tasks could be supported such as clone detection, code modeling and categorisation.

The remainder of this paper is organised as follows. In the next section we provide a brief overview of the main components in RASCAL. This is followed by a detail explanation of the recommendation algorithm and a comparison of a number of different Bayesian techniques in section 3. Section 4 presents experimental results with discussion. Related works are reviewed in section 5. Finally we discuss how RASCAL can be extended and draw general conclusions in section 6.

2. System Overview

RASCAL is currently implemented as a plugin for the Eclipse IDE. As a developer is writing code, RASCAL monitors the methods currently invoked and uses this information to recommend a candidate set of methods to the developer. Recommendations are then presented to the developer in the recommendations view at the bottom right hand corner of the IDE window. At present, RASCAL recommends methods from the Swing and AWT libraries. Below we describe the main components of RASCAL, as shown in figure 1.

We produce personalised recommendations for each individual **Developer**. When producing a recommendation, we only consider the content of the current active method which this developer is coding. In recommender systems, it is common terminology to refer to the user for whom the recommendation is being sought as the active user; likewise here we will refer to the active developer or the active method that a developer is coding. The **Code Repository** contains code from previous projects, external libraries, open-source projects etc; in our work we used the Sourceforge [8] repository. This repository will be continually updated as new classes/systems are developed. From such a repository, we can extract information about what reusable library methods exist and also knowledge about how these are used. We produce an **Information Retrieval Model** by mining the code repository; the actual information retrieval model used can vary as discussed in section 3.2. This model will need to be created once initially and subsequently when a new piece of source code is added to the repository. We extract information from the repository using the *Bytecode Engineering Library* [1].

Finally there will be a **Recommender Agent** for each individual developer; this agent actively monitors the method that the developer is coding. The agent then uses the in-

formation retrieval model to establish a set of source codes that are most similar to the code currently being written by the developer and following this, a set of ordered library methods is recommended to the active developer. The recommendation set is produced based on the similar source codes; we explain the recommendation technique in full in the following section.

3. Recommendations

3.1 Collaborative Filtering

The goal of a Collaborative Filtering (CF) algorithm is to suggest new items or predict the utility of a certain item for a particular user based on the user’s previous preference and the opinions of other like-minded users [25]. CF systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items and will likely agree on future items. CF algorithms are used in mainstream recommender systems like *Amazon*. In our work we use CF to recommend a set of library methods to a developer.

For clarity we describe three terms, specific to this context, that are common terminology in recommender literature. An **item** refers to a reusable library method. We wish to predict a developers preference for an item. A **user** is a Java method in our source code repository. The active user can be considered as the method currently being written or indeed the actual developer of that method. Finally a **vote** represents a users’ preference for a particular item. In this context, a vote is simply an invocation count for a particular library method.

3.1.1 Recommendation Algorithm

Breese et al. [3] identify two classes of CF algorithms, namely Memory-Based and Model-Based. In a memory-based approach, a prediction for the active user is based on the opinions of like-minded users. In contrast, model-based CF first learns a descriptive model of user preferences and then uses it for predicting ratings. Employing a memory-based algorithm, vote v_{ij} corresponds to the vote by user i for item j (invocation count in this work). The mean vote for user i is:

$$\bar{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j} \quad (1)$$

where I_i is the set of items the user i has voted on. The predicted vote using CF for the active user a on item j , cf_{aj} , is a weighted sum of the votes of the other similar users:

$$cf_{aj} = \bar{v}_a + N \sum_{i \in kNN} sim(a, i) (v_{i,j} - \bar{v}_i) \quad (2)$$

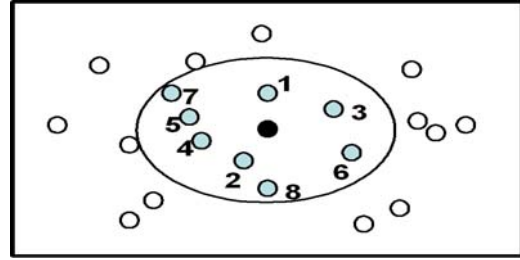


Figure 2: Illustration of the kNN formation. Here we look for the active methods’ $k=8$ most similar source codes.

where weight $sim(a, i)$ represents the correlation or similarity between the current user a and each user i . kNN is the set of k nearest neighbours to the current user, as illustrated in figure 2. A neighbour is a user who has a high similarity value $sim(a, i)$ with the current user. The set of neighbours is sorted in descending order of weight. For experiments we used a value of $k = 10$. N is the normalising factor such that the absolute values of the weights’ sum to unity. From equation 2 we can now predict a users’ vote for any item. In the context of this work, we can now predict a developers’ vote for any library method assuming that there exists at least one snippet of code in the code repository that has used the particular library method. Library methods are ranked based on their predicted vote and the top n methods are recommended to the developer. In our experiments, we use a value of $n = 7$.

Central to CF is the ability to determine a set of users who are most relevant or similar to the active user for whom the recommendation is being sought, $sim(a, i)$. We want to effectively discover source codes in our repository that are most similar to the code currently being written. The Information Retrieval (IR) model chosen will have a direct impact on which users are deemed relevant and which are not, and thus ultimately impacts the recommendation set. Baeza-Yates and Ribeiro-Neto [2] identify three basic retrieval models; boolean, vector/statistical and probabilistic.

In previous works we have investigated the suitability of vector approaches in the software component recommendation domain [4]; namely we looked at the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) and found VSM to produce the best results. Here we investigate how effective probabilistic approaches are at ranking source code based on similarity. This is equivalent to classification in machine learning; however, we are attempting to classify the top n pieces of code that are most similar to the active method being written. Typically statistical approaches are used for memory-based algorithms while probabilistic techniques are used with model-based algorithms. In this work, we employ a hybrid approach akin to the work of [23]. Like the model-based technique, we construct a Bayesian net-

work though we treat each method as a unique cluster and therefore when making a prediction, we need to consider all methods in the code repository.

3.2 Bayesian Network Classifiers

A **Naïve-Bayes** BN [7, 16] is a simple structure that has the classification node as a parent of all other attribute nodes. Naïve-Bayes is based on the assumption that the attributes values are independent of each other given the class C . In the context of this work, the classification node would represent a particular piece of code from the code repository, whereas an attribute node represents each reusable library method that can be invoked. The conditional probability of each attribute given the class C is learnt from training data. Classification is then done by applying Bayes rule to compute the probability of C given a particular instance of attributes and then predicting the class with the highest posterior probability. In this work, we wish to determine the top kNN pieces of code that are most similar to the query instances.

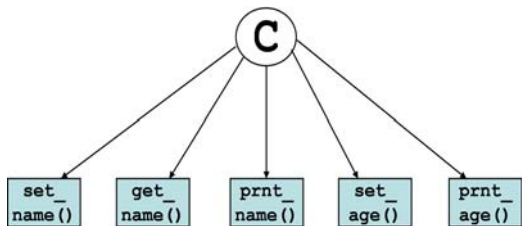


Figure 3: Naive-Bayes Network

Despite the Naïve assumption of probabilistic independence between attributes, Naïve-Bayes classifiers in general work reasonably well; indeed they have been shown to outperform BN [9]. This is surprising given that the attribute assumption rarely holds in real world examples. In our domain we might expect that there would be a relationship between at least some of the methods in the reusable library; we investigate if the Naïve-Bayes BN can effectively classify source code whilst ignoring such relationships. Figure 3 displays an example of Naïve-Bayes Network.

A general **Bayesian Network** (BN) [22] is a much more powerful representation of probabilistic dependencies over a set of random attributes; a BN can effectively model the complex dependencies that exist in most real world problems. More formally, a BN is a directed acyclic graph with nodes representing attributes and arcs representing dependence between relations among the attributes. Probabilistic parameters are encoded in a set of tables (Conditional Probability Tables), one for each attribute node, in the form of logical conditional distributions of a attribute given its parents. Using the independence statements encoded in the net-

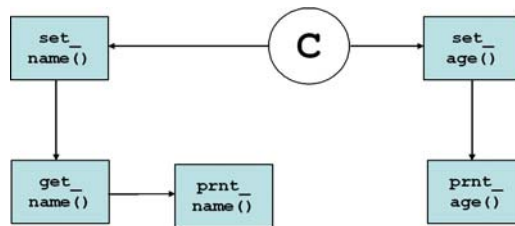


Figure 4: Bayesian Network

work, the joint distribution is uniquely determined by these logical conditional distributions. Figure 4 displays an example of a general BN; unlike Naïve-Bayes the classification node is treated the same as the attribute nodes. As is suggested by Cheng and Greiner [5], this lack of distinction between the classification and attributes nodes is not always desirable in certain domains and thus we introduce Bayesian Networks Augmented Naïve-Bayes shortly.

Learning a BN based classifier is a computationally challenging problem; if the network is unrestricted then it is a NP-hard problem. We need to find a network that best matches the entire instances in the training data. Using a scoring function we need to evaluate each learnt network against the training data and determine the optimal network.

Several authors have proposed a compromise between the computationally expensive Bayesian network model and the over-simplified Naïve-Bayes approach. The desire is to merge the ability of BN to model attribute dependence with the simplicity and efficiency of Naïve-Bayes BN. Friedman et al. [9] define such structures as Augmented Naïve Bayesian Networks. Each attribute must have a class attribute as a parent and each attribute may have one other parent [15]. From figure 5, it can be seen that it is now possible to model dependency between attributes whilst maintaining the simplicity of the Naïve-Bayes BN. In general, as stated earlier, learning an unrestricted network is a NP-Hard problem. Friedman et al. [9] deal with this by restricting the network to a tree topology; the result is known as a **Tree Augmented Naïve-Bayes** (TAN) as is specifically shown in figure 5. There is an arc from `getName()` to `setName()` and thus these two attributes are not independent given the class.

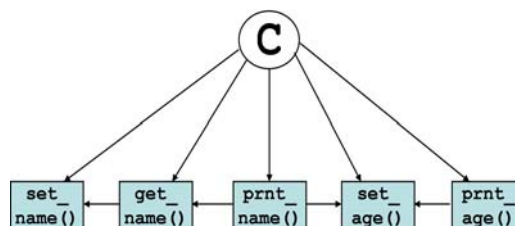


Figure 5: TAN

Keogh and Pazzani [15] present a similar tree augmented network but unlike TAN, which adds $N - 1$ arcs (where N is the number of attributes), they add any number of arcs up to $N - 1$. An arc is only added if it improves accuracy. This same approach is defined by Sacha [24] as a **Forest-Augmented Network** (FAN), as the augmenting arcs form a forest of attributes (or a collection of trees); this is illustrated in figure 6.

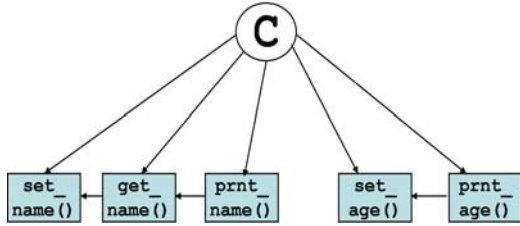


Figure 6: FAN

The final BN we consider is the **Bayesian Network Augmented Naïve-Bayes** (BAN). This extends TAN by allowing attributes to form an arbitrary graph, rather than just a tree, as is shown in figure 7. This is similar to the original general BN but in this case the classification node is treated differently from the rest of the attribute nodes. It is hoped that the BAN will more richly model relationships between attributes but this will likely come at a computational cost. A more detailed comparison of Bayesian networks can be found in [5].

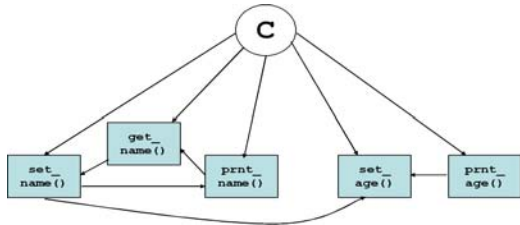


Figure 7: BAN

Excluding general BN’s and FAN’s, all the above networks were constructed using the popular WEKA [27] machine learning tool. We used a repeated hill climbing searching algorithm (maximum of 5 runs) and the BDeu scoring function. As general BN’s do not distinguish between class and attribute nodes, we decided to implement the more efficient BAN instead; the number of parent nodes was limited to 4. For the FAN implementation we used the Java Bayesian Network Classifier (JBNC) toolkit [14]. All training data was normalised and discretised to have 3 values; for example, the method `setName()` may be invoked either between 0 and .33 times, between 0.34 and 0.66 times or finally between 0.67 to 1 times.

4 Experiments

4.1 Dataset

In these preliminary experiments, we used relatively small datasets. We produced almost 6000 recommendations from approximately 350 methods mined from Sourceforge [8]. Recommendations were produced solely at the method level and not the class level as in previous work [18]. Further to this, each method had on average 16 invocations. Recommendations were made for both the SWING and AWT libraries; in total there was 697 Swing and AWT library methods that were invoked at least once in our code repository. Although the data is small for this domain, 697 instances and 350 classes is comparatively large with experiments carried out in machine learning literature. Since we have the completed source code, we can automatically evaluate recommendations for a piece of code by checking whether the recommended method was called subsequently.

For each of the 350 methods, several recommendations were made. For example, if a fully developed method had 10 Swing invocations, then we removed the 10th invocation from that method and a recommendation set was produced for the developer based on the preceding 9 invocations. Following this recommendation, the 9th invocation was removed and a new recommendation set was formed based on the preceding 8 invocations. This process was continued until just 1 invocation remained. Each recommendation set contained a maximum of 7 items.

4.2 Evaluation

Precision and Recall are the most popular metrics for evaluating information retrieval systems. Precision is defined as the ratio of relevant recommended items to the total number of items recommended; $P = n_{rs}/n_s$, where n_{rs} is the number of relevant items selected and n_s is the number of items selected. This represents the probability that a selected library method is relevant. A library method is deemed relevant if it is used by the developer for whom the recommendation is being sought. Recall is defined as the ratio of relevant items selected to the total number of relevant items; $R = n_{rs}/n_r$, where n_{rs} is the number of relevant items selected and n_r is the number of relevant items. This represents the probability that a relevant library method will be selected.

It is particularly important that RASCAL recommends methods in a relevant order i.e. the invocation order. We will evaluate this using a simple binary Next Recommended (NR) metric; $NR = 1$ if we successfully predict or recommend the next method a developer will use, otherwise $NR = 0$. In these investigative experiments we focused

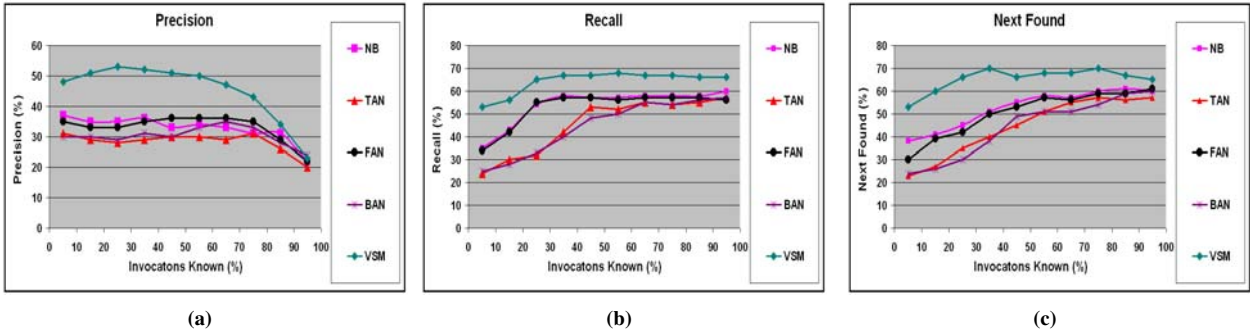


Figure 8: (a) Precision (b) Recall (c) Next Found

solely on the above 3 metrics whilst ignoring computational complexity.

4.3 Results

All results are displayed as a percentage value. A baseline result is included; this was produced using the Vector Space Model (VSM) as detailed in previous work [4]. From figure 8, it is immediately identifiable that the VSM baseline result produced the best results in general. While this may not have been the desired outcome, there is still insight to be gained from the results.

Precision is displayed in figure 8(a). VSM vastly outperforms all of the Bayesian techniques; for example, the average FAN precision is 33% which compares poorly with 45% when using VSM. Recall is shown in figure 8(b); again VSM outperforms all other techniques. We notice that Naïve-Bayes (NB) and the Forest-Augmented Network (FAN) produce similar results and that these are both marginally better than the Tree Augmented Network (TAN) and the BN Augmented Naïve-Bayes (BAN).

The next found metric is displayed in figure 8(c). Using NB, there is a 48% likelihood that RASCAL would be able to correctly predict the next library method that a developer would invoke; such a prediction would provide significant help to a developer who was unfamiliar with a particular library. In general, this is an encouraging result yet it is relatively poor when compared with the VSM 64% average.

4.4 Discussion

From this exploratory research on using Bayesian networks to recommend library methods, we can make some interesting observations. Firstly we discover that FAN and NB produce similar results for all metrics. This would suggest that the FAN added very few links as these did not improve classification. We also notice the similarities between TAN and BAN; again this would suggest that the

BAN was very similar to the TAN created and that there is no benefit to having multiple parents. In the context of this work, this can be interpreted as there being very few relationships between library methods and hence a Naïve-Bayes network will produce better recommendations. Further work is needed to verify this. In addition to this, further investigation is needed in the area of searching and scoring techniques to ensure they are ideally suited to this domain.

Generally, we notice two different trends in precision and recall. Precision tends to decrease as we know more information about the active method while recall tends to increase. This result perhaps requires clarification. Consider a developer who invokes in total 10 methods. When we make a recommendation for that developer when they have only used 1 method, there is a set of 9 possible methods to recall. The chances of recalling all relevant methods is quite low and hence the recall result is low in earlier recommendations. However, when this developer has used 9 methods and there is only 1 possible method to recall, then the chances of this method being in the recommendation set is quite high. In contrast, the more invocations the developer has made, the fewer there are to correctly recommend and hence precision decreases in latter recommendations.

5 Related Work

Traditional retrieval schemes focused generally on techniques such as *Keyword Search* and *Signature Matching* [19]. More recently several *Semantic-Based* retrieval tools have been proposed [26, 10]; these allow a developer to specify queries using natural languages. Unlike traditional retrieval, the domain information, developer context and component relations are considered. Empirical results indicate that these tools are superior to traditional approaches.

ComponentRank [13] is a promising component retrieval technique which is useful for locating reusable components. Similar to *Google* [21], this approach ranks components

based on analysing use relations among the components and propagating the significance of a component through the use relations. Preliminary results indicate that this technique is effective in giving a high rank to stable general components which are likely to be highly reusable and a lower rank to non-standard specialised components. Similarly, Hummer and Atkinson [12] have carried out a general study on using the web as a reuse repository; they evaluate several search engines such as *Google*, *Yahoo* and *Koders*. They identify some of the advantages of web based approaches such as scalability and efficiency but also note limitations such as security, legal concerns and implicit classes.

The use of software agents for supporting and assisting library browsing have been proposed by Drummond et al. [6]. An active agent attempts to learn the component which the developer is looking for by monitoring the developers' normal browsing actions. Based on experimental results, 40% of the time the agent identified the developers' search goal before the developer reached the goal. By providing non intrusive advice that accelerates the search, this work is intended to complement rather than replace browsing.

A major limitation with all of the retrieval techniques above is that the developer must initiate the search process. However, in reality developers are not aware of all available components or methods in a library. If they believe a reusable component for a particular task does not exist then they are less likely to search the component repository; none of the above schemes attempt to address this important issue. Thus to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with component delivery/recommendation.

Ye and Fischer [28] identify the cognitive and social challenges faced by software developers who reuse and also present a tool named *CodeBroker* which address many of these challenges. *CodeBroker* infers the need for components and pro-actively recommends components, with examples, that match the inferred needs. The need for a component is inferred by monitoring developer activities, in particular developer comments and method signature. This solution greatly improves on previous approaches but the technique is not ideal. Reusable components in the repository must be sufficiently commented to allow matching and developers must also actively and correctly comment their code which currently they may not do. Notably, Ye and Fischer remark that browsing and searching are passive mechanisms because they become only useful when a developer decides to make a reuse attempt by knowing or anticipating the existence of certain components.

Mandelin et al. [17] present an intelligent tool for understanding and navigating the API of a particular reuse library. They suggest developers often know the objects they would like to use but are unaware of how to write the source

code to get the object; for example a developer may wish to create a *IFile* object from a *ASTNode* but may not be aware of the code needed to do this. They provide a tool named *PROSPECTOR* which can automatically assist a developer to better understand the library API by providing code snippets relevant to the current task; for example, how to convert between different data representation or traversing object schemas.

Another notable tool for finding code examples is Strathcona [11]. The tool is used to find source code in an example repository by matching the code a developer is currently writing. Similarity is based on multiple structural matching heuristics, such as examining inheritance relationships, method calls, and class instantiations. These measures are applied to the code currently being written by the developer and matched examples from the repository are retrieved and recommended.

Our work is similar to a number of the techniques mentioned above. Like *CodeBroker* [28], our goal is to recommend a set of candidate software components to a developer; however, our recommendations are not based on the developers' comments/method signature. In contrast we produce recommendations using CF which is similar to the example based techniques of Holmes and Murphy [11]. Like the *PROSPECTIVE* tool, we are interested in increasing and supporting library reuse though we are attempting to predict in advance what a developer is attempting to code. Like Drummond et al. [6] we use an active agent to monitor the current developer though we are concerned with pro-actively recommending suitable reusable methods as opposed to assisting the search process.

6 Conclusions

We have presented a solution that automatically facilitates knowledge sharing within a community. We have shown that just as people can be clustered in terms of their preferences for various items, Java source code may also be clustered based on the library methods invoked. We note the importance of correctly identify the optimal technique for clustering source code; we investigated a number of Bayesian techniques and compared these with our VSM statistical baseline result.

In this work, we discovered conclusively that Bayesian Networks are less useful at clustering source codes than VSM and ultimately have a negative effect of recommendation performance. Further and larger experimentation is needed to generalise this finding though; in particular we need to evaluate more search and scoring techniques. Bayesian techniques do still offer promising opportunities for us; for example, modeling relationships between library methods, classification or clustering of library methods as opposed to classifying entire source codes as is presently

done and finally applying the discussed Bayesian techniques to pure model-based CF.

Our recommendation scheme addresses various shortcomings of previous solutions to the library retrieval problem; RASCAL considers the developer context and problem domain but uniquely does not place any additional requirements on existing library components or developers. Unlike many typical reuse tools, RASCAL is proactive and constantly suggests library methods to reuse.

Recommender systems are a powerful technology that can cheaply extract knowledge for a software company from its code repositories and then share this knowledge to the benefit of future developments. We have demonstrated that RASCAL offers real promise for allowing developers discover and easily access reusable library components but that care needs to be taken when choosing the clustering technique.

7 Acknowledgements

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology (IRCSET) under grant RCS/2003/127.

References

- [1] Apache. Bytecode engineering library (2002-2003). <http://jakarta.apache.org/bcel>. 2003.
- [2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
- [3] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 43–52, 1998.
- [4] F. M. Carey, M. O. Cinnéide, and N. Kushmerick. Recommending library methods: An evaluation of the vector space model (vsm) and latent semantic indexing (lsi). In *9th International Conference on Software Reuse*, Italy, 2006.
- [5] J. Cheng and R. Greiner. Comparing bayesian network classifiers. In *Proceedings of UAI*, pages 101–108.
- [6] C. G. Drummond, D. Ionescu, and R. C. Holte. A learning agent that assists the browsing of software libraries. *IEEE Trans. Softw. Eng.*, 26(12):1179–1196, 2000.
- [7] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1973.
- [8] J. Ebert. Storm - a user story tool. <http://xpstorm.sourceforge.net>. 2002.
- [9] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.
- [10] M. Girardi and B. Ibrahim. Using english to retrieve software. *Journals of Systems and Software*, 30(3):249, 1995.
- [11] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.
- [12] O. Hummel and C. Atkinson. Using the web as a reuse repository. In *Proceedings of the 9th International Conference on Software Reuse*, pages 298–311. Springer, 2006.
- [13] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, pages 14–24. IEEE Computer Society, 2003.
- [14] J.P.Sacha. Java bayesian network classifier (jbnc) toolkit. <http://jbnc.sourceforge.net>. 2004.
- [15] E. Keogh and M. Pazzani. Learning augmented bayesian classifiers: A comparison of distribution-based and classification-based approaches, 1999.
- [16] P. Langley, W. Iba, and K. Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.
- [17] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. *SIGPLAN Notices*, 40(6):48–61, 2005.
- [18] F. McCarey, M. O. Cinnéide, and N. Kushmerick. Knowledge reuse for software reuse. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, July 2005.
- [19] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.
- [20] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA. IEEE Computer Society.
- [21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.
- [22] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [23] D. M. Pennock, E. Horvitz, S. Lawrence, and C. L. Giles. Collaborative filtering by personality diagnosis: A hybrid memory and model-based approach. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 473–480, CA, USA, 2000.
- [24] J. Sacha. *New synthesis of Bayesian network classifiers and interpretation of cardiac SPECT images*. Ph.d. dissertation, University of Toledo, 1999.
- [25] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *World Wide Web*, pages 285–295, 2001.
- [26] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.
- [27] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, 2005.
- [28] Y. Ye and G. Fischer. Reuse-conducive development environments. *International Journal of Automated Software Engineering*, 12:199–235, 2005.
- [29] K. Yongbeom and E. Stohr. Software reuse: Survey and research directions. *Management Information Systems*, 14(4):113–147, Spring 1998.

Assisting Concept Assignment using Probabilistic Classification and Cognitive Mapping

Brendan Cleary & Chris Exton

*Department of Computer Science and Information Systems,
University of Limerick,
Ireland.*

Brendan.Cleary@ul.ie, Chris.Exton@ul.ie

Abstract

The problem of concept assignment, that is, the problem of mapping human oriented concepts to elements in the code base of a system under study, and approaches which facilitate concept assignment can be considered as central to assisting software engineers in comprehending the unfamiliar systems they encounter. This paper presents a technique called cognitive assignment that attempts to capture what expert engineers know about the systems they work with and uses that information to generate classifiers that are used to implement a ranked search over a set of software elements.

1. Introduction

When a software engineer encounters an unfamiliar system for the first time, that engineer is tasked with understanding some or all of that system before they are able to make any meaningful contribution to its development or maintenance. While this problem is readily evident in cases of novice engineers joining existing projects [1] it also applies to experienced engineers moving between projects or in cases where a system acquired from one organization needs to be developed or maintained by another.

Tools which assist software comprehension are an integral part of the solution to this “ramp-up” problem, however while software comprehension is widely recognized as one of the pervasive problems of software engineering and while many authors have tried to establish models of how software comprehension occurs [2] [3], few authors have attempted to define what it means for a software engineer to comprehend a software system. Good recognising this deficiency ventures a definition of software comprehension in [4] which

can be considered as being characteristic of other authors attempts [5] [6] in that it establishes comprehension as a process which sees the engineer use information drawn from different sources to form a model of the software which is then used by the engineer in answering questions about the system in the context of performing some task.

Biggerstaff in defining what it is for an engineer to comprehend a software system takes a different perspective, one which de-emphasises models of comprehension and instead looks at what is required for an engineer to be said to comprehend a system [7]. This definition describes software comprehension in terms of an engineer’s ability to communicate intelligently in human oriented terms about a systems implementation. This categorisation of software comprehension rests on two different expressions of “computational intent” and the ability of the software engineer to associate concepts appearing in one description of intent with the concepts in another. Intent is what developers intend when they write software [8]. Different descriptions of intent are separated by constraints on the sets of concepts expressible using the language in which they are described. These constraints constitute a “conceptual gap” [9] between different descriptions or domains of intent. In considering software comprehension we are usually concerned with two descriptions of intent, one described using a human language (problem domain) another using a programming language (solution domain). While a systems implementation may imply the intent that led to its development it is not expressed explicitly rather it is expressed using terms defined by the implementation technologies rather than in terms that appear naturally in the intent [10]. Biggerstaff describes the problem of associating concepts between these different descriptions of intent or domains as the concept assignment problem [11].

Existing approaches which explicitly attempt to assist engineers to bridge this conceptual gap such as tool assisted, lexical, statistical and dynamic concept assignment approaches tend to rely on the parsing of solution domain artefacts (source code) to identify elements of the code base which are then inferred to be related to the implementation of some set of concepts from the problem domain. In this paper we present a complementary concept assignment approach based on the combination of a quantitative text analysis technique called cognitive mapping [12] and probabilistic classification [13].

In section 2 we look at related work from various fields that attempt to alleviate the concept assignment problem. In section 3 we describe our proposed technique the effectiveness of which is tested by an experiment described in section 4 and analysed in section 5. Finally in section 6 we describe limitations to our technique and evaluation and in section 7 we describe our conclusions and future work.

2. Related Work

Given the scope of the concept assignment problem, many techniques and tools from the software visualisation, comprehension, reengineering and even requirements engineering communities could be classified as attempting to tackle the concept assignment problem, here we will briefly examine a small subset of those that explicitly set out to do so.

Dynamic software analysis techniques such as software reconnaissance [14] or formal concept analysis [15], focus on localising concepts that are expressible either through test cases or through navigation of control and data flow. Unfortunately while a systems implementation may imply the intent that led to its development, the intent is not expressed explicitly in that implementation [10]. As such these techniques are only able to localise concepts which are expressible as test cases. While this is a limitation, in cases where there exists no system expert or documentation, they can be of great benefit in assisting engineers understand these dark systems.

Other significant areas of related work which attempt to capture and describe the relationship between problem domain concepts and the code base include tool assisted techniques such as the Concern Manipulation Environment (CME) [16] and FEAT [17] which allow engineers to explicitly describe and record associations between software elements and user defined concerns, and artefact recommender systems such as Hipikat [18]

which suggest pertinent artefacts (both code and documentation) to engineers as they engage in an understanding task. While we see these tools as closely related and complimentary to our approach; cognitive assignment differs in that it incorporates problem domain information not present in the code, captured from a system expert or experts, to assist the choices novice¹ users make when mapping problem domain concepts to elements of the code base.

Another significant area of related work are the studies into software engineer work practices carried out by Singer and Lethbridge in the mid to late 90's [19]. Using a set of field research techniques including; interview, shadowing and questionnaires which the authors collectively term software anthropology [20], Singer and Lethbridge performed a series of experiments in which they studied the work practise of software engineers as they engaged in their day to day activities. Their findings across all three studies demonstrate that search was overwhelmingly the dominant activity engaged in by the software engineers they observed. In the longitudinal study of a novice engineer, searching and looking at the source accounted for over 50% of events observed by the authors. In a second study, while editing and debugging grew in importance, searching still accounted for a significant proportion of events observed. Finally a study of tool usage statistics revealed that close to 50% of the calls made by engineers across the company were calls to grep-like search programs.

In this light of the importance of search to software understanding Marcus and Maletic [21] describe the application of an information retrieval technique called Latent Semantic Indexing (LSI) in recovering traceability links between documentation and source code. Marcus et al. expand on this work applying LSI directly to the concept location problem in [22] where they build an index of terms from identifiers and comments in the source code which are then used to localise a user specified query to a set of functions. While our approach coincides with Marcus et al's approach in terms of intent and granularity of localisation, we differ first in that our index is derived not from the source code but from software engineers with expertise in the system under study through cognitive mapping and second in that we use a different classifier to construct the mapping between a user query and the code base.

¹ We use the term novice to indicate software engineers encountering an unfamiliar SUS, these engineers may or may not have experience with other systems.

3. Cognitive Assignment

The cognitive assignment technique consists of 2 phases; a cognitive map derivation phase (performed once per each system-expert pair) and a concept assignment phase (performed each time a novice generates a query). The cognitive map derivation phase first semi-automatically derives a cognitive map from an expert software engineer related to a System Under Study (SUS) by analysing texts related to the SUS authored by the expert, such as design documentation, bug reports, or transcripts of interviews with that expert. The concept assignment phase then, each time the novice specifies a query, generates a probabilistic classifier based on a subset of the concepts and relationships in the expert's cognitive map. This subset is defined in terms of the set of concepts specified in the novices query. The generated classifier is then used to classify elements of the SUS code base according to their probable relation to concepts in cognitive map. These classifications or rankings are then displayed to the novice for their investigation through a search results interface integrated into the Eclipse IDE. The next section describes the theory behind cognitive mapping and cognitive maps.

3.1. Cognitive Mapping

A mental model is the model people have of themselves, others, the environment, and the things with which they interact, formed through experience, training and instruction [23]. Based on the assumption that language and knowledge can be modelled as networks or maps of words and the relations between them [24], texts can be thought of as containing a portion of the author's mental model at the time the text was created [12]. Working under the assumption that the meaning of a text does not result from single words but from the co-occurrence of different words [25], cognitive mapping is a quantitative text analysis technique that systematically extracts and analyses the links between words in a text in order to model the authors mental or cognitive map as networks of words [26] [27]. This map is then hypothesised to approximate a portion of the mental model of the texts author at the time the text was composed [28].

While current general purpose programming languages do not allow for the direct expression of programmer intent [29] [10], software engineers have long used other software artefacts such as requirements, architectural and design documentation and more recently email, bug tracking databases and wikis to express concerns which cannot be expressed directly in the source code. Analysing these texts using cognitive mapping

allows us to extract and make explicit the portion of the software engineer's mental model relative the system under study expressed within as maps of concepts, thus capturing and making explicit some of the original intent of the engineer. These maps can then be bound, using a classifier function, to elements in the code base of the SUS.

In [12] Carley and Palmquist present a methodology for extracting, representing and analysing cognitive maps from a corpus of texts consisting of 4 phases;

- A concept set definition phase where the set of concepts which the map is to be constructed from are identified using text pre-processing techniques which eliminate all words from the texts but those which are considered by the researcher to be important in answering the research questions.
- A relationship type definition phase that identifies the relationship types that can exist between concepts in the map, again the relationship types are determined by the researcher.
- A map construction phase where a computer-assisted coding of texts is performed using the identified concepts and relationship types. A set of statements is constructed using a windowing technique from which a map is created based on the union of the set of statements.
- Finally a map analyses phase renders the resultant maps for analysis by the researcher.

Applying cognitive mapping to texts produced by software engineers for the purposes of facilitating concept assignment requires that we customise the method presented above so that it can be applied in a production software development environment. This requires that we automate as much of the process as possible while at the same time attempting to maintain the qualitative nature of the cognitive mapping process. As such we propose to operationalize the cognitive mapping procedure of Carley and Palmquist into one consisting of 2 phases;

- A semi-automated concept set definition phase which identifies a set of concepts from a corpus of text segments using a combined manual content analysis and semi-automatic text pre-processing approach.
- A completely automated map construction phase which uses the set of concepts identified in the concept set definition phase as the basis on which conceptual maps are constructed using a windowing based approach, which creates statements be-

tween concepts in text segments which co-occur within the window.

The next section describes how we construct classifier functions from subsets of concept and relationships captured in a cognitive map and how we use those classifiers to generate rankings for individual software elements.

3.2. Bayesian Classification

Classification is a basic task in data analysis and pattern recognition that requires the construction of a classifier, that is, a function that assigns a class label to instances described by a set of attributes [30]. Applied to text classification, a naïve Bayesian classifier function, given a set of training texts and associated example classifications, determines the probability of a given term (attribute) occurring for each of the given classifications over the set of training texts. This model of conditional term probability can then be used determine the classification of an unseen text based on the product of the probabilities of the set of terms contained in the unseen text. Term or attribute probability is usually calculated based on frequency of occurrence, for example Mitchell divides the frequency of occurrence of a term in the training set by the sum of the total number of distinct word positions in the training data for the classification and the total number of distinct words in the training data [13].

In relation to the concept assignment problem a probabilistic model, based on naïve Bayesian classification, has already been used by Antoniol et al [31] for recovering traceability links between code and documentation. Here the authors used unigram estimation based on term frequency to create links that describe the similarity between elements of the code base (object-orientated classes) and high level system documentation. The authors use a stochastic language model based on identifiers found in the source code elements to calculate the set of conditional probabilities between a given source code element and the set of system documents. Naïve Bayesian classification has also been used to assist in automatically assigning bug reports to engineers with specialist knowledge [32]. Here the authors use an existing database of assigned bugs to learn a naïve Bayesian classifier that can automatically assign or classify unseen bug reports to particular engineers based on previous classifications of bugs that were made.

While being one of the most effective classifiers [30], to make the calculation of the set of conditional prob-

abilities computationally tractable, the naïve Bayesian classifier has to make a strong independence assumption that all attributes are conditionally independent given the value of the class attribute. That is, given attributes A and B and a class C , $\Pr(A|B,C) = \Pr(A|C)$ for all values of A, B and C , whenever $\Pr(C) > 0$. In text classification this independence assumption means that the order or sequence of occurrence of words in a subject text is not taken into consideration in its classification. As such naïve Bayesian text classifiers are sometimes described as treating texts as “bags of words”.

While naïve Bayes classifiers have been shown to be remarkably efficient given their simple structure, the independence assumption on which they are based is clearly not always valid. This observation lead some researchers to relax the independence assumption in an attempt to create better performing classifiers that maintain the desirable computational characteristics of naïve Bayesian while incorporating more information about dependencies between attributes.

In [30] the authors discuss the modification of a naïve Bayes classifier with augmenting edges between attributes that describe the dependencies between those attributes which are then taken into consideration when used as a classifier, thus relaxing the independence assumption of the naïve Bayes. However in order to maintain the naïve Bayes’s computationally tractable performance the authors refrain from developing augmenting edges between each pair of attributes. Instead by applying a maximum spanning tree algorithm [33] over the attribute set they are able to construct the optimal set of augmenting edges in polynomial time.

3.3. Cognitive Assignment

Our cognitive assignment procedure uses a probabilistic model, based a tree augmented Bayesian classifier formed from a subset of an experts cognitive map, to classify elements of a SUS code base in terms of how related they are to a concept set (classification) defined by the novice engineer.

Given a cognitive map M defined by an expert for a system under study S , the procedure for constructing the classifier and applying it to classification of a set of elements is as follows;

1. The novice engineer, engaged in assigning a concept C , to a set of source code elements E in S , defines a subset of the experts cognitive map

- m , consisting of a set of concepts from M which the novice considers related to the concept or class C which she is attempting to localise. We call the subset, m , a concern map.
- Given the concern map m we construct a tree augmented classifier X_m by computing a mutual information function over the set of pairs of concepts in m based on their individual and co-occurrence frequencies derived from the original texts and the cognitive map, respectively. Then using this score we annotate the edges between the pairs of concepts and derive a maximum spanning tree over the set of concepts in m .
 - We then transform the resulting undirected tree into a directed one by picking a root concept and setting the direction of all edges to be outward from it.
 - This classifier, X_m , is then used to classify the set of source code elements E according to how related those elements are to the concept C as defined by the novice engineer in m . This relationship is established based on the occurrence of concepts from m in the text of the source code elements, which includes both executable and non-executable statements.

This process is repeated each time the novice engineer generates a query set of concepts using the tool support provided in the cognitive assignment Eclipse plug-in. The cognitive assignment plug-in [34] is an Eclipse plug-in that implements the second phase of the cognitive assignment technique described above to allow an engineer encountering an unfamiliar system to construct and record a set of associations between problem domain concepts captured by a system expert in a cognitive map and the elements of the SUS code base that comprise that system. The next section describes an experiment in which we assess the performance of the cognitive assignment Eclipse plug-in in generating correctly ranked element sets.

4. Evaluation

To evaluate our proposed technique, we conducted a small lab based experiment with 4 participants to quantitatively assess the performance of our cognitive assignment Eclipse plug-in over 4 tasks in terms of precision and recall versus sets of elements defined by a system expert. A cognitive map was also defined for the SUS in the experiment using the procedure described in section 3.1. Both the expert element set and

the cognitive map we defined prior to the experiment by the primary author.

4.1. Case Study System

The experiment was performed over the CHVIE software visualisation tools framework [35]. The CHIVE has been employed in the implementation of several software understanding tools [36] and has been in development for over 3 years. The CHIVE core, the framework itself, consists of 7 packages, 25 classes and over 15 KLOC of Java. Finally between the client applications and the framework there is over 40,000 words of academic and technical text documenting CHIVE and its client applications. We chose the CHIVE framework as the basis of this case study because it constitutes a non trivial system with which the authors of this paper were intimately familiar but which the participants of the study were not and finally because the source code of CHIVE is also open source.

4.2. Participant Profile

The 4 participants selected for this study were post-graduate students, with on average 6 months of academic Java development experience and 3 years of academic development experience with other object oriented languages. The participants also had on average 3 months commercial Java development experience and over 8 months commercial development experience with other object orientated languages.

4.3. Experiment Procedure

Prior to the experiment each participant was briefed on the experiments objectives and protocol. Next the participant received training in the use of the plug-in and an introduction to using Eclipse. The participants then received a 10 minute introduction to the system against which the experiment was run. Next the participants were presented with the tasks which they were to perform in series during the experiment. For each task the participants were given 5 minutes to read the description and ask the experiment supervisor questions on the description. They were then be asked to (using the cognitive assignment plug-in) identify elements² of the source of the system under study which they thought were important to the concept/task under investigation. When the participant had completed all tasks they were thanked for their contribution, debriefed and given the opportunity to review the data collected.

² For this study we limit the element of localisation (source code unit) to the Java method; however our technique is applicable to any unit of decomposition.

4.4. Task Types

The participants were asked to complete 4 tasks, 2 concept localisation tasks, a feature request task and a bug location exercise. The tasks were each described in a paragraph of text similar to that which would be entered in a use case description, feature request or bug report. The concept location tasks required the participant to identify the elements of the system which they thought were important to the implementation of the concept as described in the given use case description. The feature request task asked the participants to identify elements that they thought either would be impacted by the proposed feature request or which could be reused in the features implementation. However for this task the users were not asked to implement the feature request. Finally the participants were asked to locate the single element that was the cause of a bug described in a bug report and demonstrated to the participant by the experiment supervisor.

5. Results & Analysis

In order to assess the performance of our technique we specified, prior to the experiment, a set of “correct” elements for each of the tasks the experiment participants would perform. These expert sets allow us to assess the performance of the cognitive assignment plug-in in generating the correct result sets. Also here we present an analysis of the lowest ranked elements investigated by the participants, this analysis helps us to empirically establish limits for the calculation of the performance of our technique and also inform future research on ranked element search in software understanding tools.

5.1. Tool Precision and Recall

Our first analysis assesses how well the cognitive assignment plug-in or more specifically, the tree augmented classifier, performed. To do this for each task we captured the final concern maps that were generated by the participants performing the experiment using the cognitive assignment plug-in. We then re-generated the set of classification probabilities produced by these concern maps. This then gave us for each task 4 sets of elements ordered by the classification function. To assess the performance of the tool we then compared these sets against the expert element set for each task.

	Task 1	Task 2	Task 3	Task 4	Average
Relevant Elements	15	9	17	1	10.5
Top 10 Total	4.75	5.75	6.5	0.5	4.375
Top 10 Recall	0.3167	0.6389	0.3824	0.5	0.45948
Top 10 Percision	0.475	0.575	0.65	0.05	0.4375
Top 20 Total	5.75	6	6.5	0.5	4.6875
Top 20 Recall	0.3833	0.6667	0.3824	0.5	0.48309
Top 20 Percision	0.575	0.6	0.65	0.05	0.46875

Table 1 - Technique Precision & Recall

We use element recall (Equation 1) to measure the number of elements correctly retrieved from the set of elements against the total number of correct elements as defined by the expert. Element precision (Equation 2) then measures the number of relevant elements retrieved against the total number of elements retrieved.

$$\text{Element Recall} = \frac{\text{Number of relevant elements reterived}}{\text{Total number of relevant elements in collection}}$$

Equation 1 Element Recall

$$\text{Element Precision} = \frac{\text{Number of relevant elements retrieved}}{\text{Total number of elements retrieved}}$$

Equation 2 Element Precision

Table 1 shows the element precision and recall achieved by the cognitive assignment plug-in, using the concern maps generated by the participants, against the expert defined element sets for the top 10 and 20 elements positions of each of the 4 tasks and the average. Here we show that our technique was able to achieve on average 45 and 43 percent recall and precision respectively when we consider the top 10 positions in the results. This rises slightly to 48 and 46 percent when we consider the top 20 positions. The cognitive assignment classifier function best performed in task 2 where we achieved precision and recall of over 60% in the top 20. While the recall in the top 20 on average was not as high as we anticipated we were satisfied with the precision rates across the first 3 tasks (task 4 had only a single correct element and so precision tends not to record the classifiers performance on this task very well).

5.2. Lowest Ranked Element Investigated

One of the risks identified by the authors prior to the experiment was the potential for participants using the cognitive assignment tool to fail to investigate all relevant classification results because of the rankings allocated.

Table 2 describes the lowest ranked element investigated by participants performing the experiment using the cognitive assignment plug-in.

Participant	Task 1	Task 2	Task 3	Task 4	Average
P1	23	38	11	9	20.25
P2	3	7	9	3	5.5
P3	14	4	9	13	10
P4	2	4	11	14	7.75
Average	10.5	13.25	10	9.75	10.875

Table 2 - Lowest Ranked Element Investigated

This analysis shows that the participants tended to only investigate those elements which were returned high in the classification results. On average the participants stayed within the top 10 results. This is an especially stark finding when we consider that 269 elements (the number of methods in the SUS) were classified and returned to the participants for each task. While these results are only preliminary we consider this a potential risk factor to the use and adoption of ranked search results to assist in concept assignment, in that if the classification function used to generate the rankings does not return the “correct” elements within the top few positions the user is likely not to investigate further down the rankings and so is likely to, initially at least, miss potentially significant elements.

6. Technique and Evaluation Limitations

Current limitations of our technique include the cognitive mapping procedure itself and the types of systems to which the cognitive assignment plug-in can be applied. The cognitive mapping procedure, originally designed as a social science research tool, can be manually intensive to implement. For this reason we are currently investigating more automatic mechanisms, which while maintaining a human in the loop, could be used for constructing simple cognitive maps in the cases where access to expert software engineers is limited. Another significant limitation of the technique is that it requires that there be a considerable amount of problem domain concepts embedded in identifiers and comments in the code. In cases where it does not hold we are investigating the use of abbreviation generator algorithms such as is presented in [37] to construct sets of candidate concepts which can be accepted in place of the problem domain concept being searched for.

Our evaluation presented here is also limited in that the size of the system under study was relatively small, 15KLOC compared to large industrial systems, as such

the performance of the cognitive mapping procedure and the cognitive assignment classification function could be a limitation when exercised over larger systems. Also the number of participants, while larger than that usually available in industrial studies, is small and so limits the generality of our results.

7. Conclusions and Future Work

We have presented here a technique for assisting concept assignment for the purposes of software understanding where engineers encounter unfamiliar systems. The cognitive assignment technique applies cognitive mapping, a quantitative text analysis technique, to texts authored by engineers familiar with existing systems. We extract from those texts the cognitive maps of the engineers related to those systems which can then be used to establish mappings between the human orientated concepts captured in the cognitive maps and elements in the system under study’s code base using a probabilistic classification function. These mappings, presented in the form of ranked search results, can then be used by engineers attempting to understand those unfamiliar systems to facilitate software comprehension.

We have implemented the cognitive assignment technique in an Eclipse plug-in and have here also presented the results of an experiment involving 4 participants where we compare the cognitive assignment plug-in’s success in generating sets of element rankings against an expert defined set. While the results of the experiment in terms of precision and recall (less than 50% on average) are not as good as we had anticipated, our immediate goal in the light for this experiment is to attempt to generalise our findings by extending this experiment to use other classifier functions and techniques such as Latent Semantic Indexing (LSI) which may demonstrate better precision and recall versus the classifier implemented here. We also wish to extend our evaluation to investigate the impact that ranked search results have on the decisions that novice software engineers make when engaged in concept assignment.

8. References

- [1] D. Cubranic and G. C. Murphy, "The ramp-up challenge in open-source software projects," presented at Workshop on Open-Source Software, held as part of the IEEE/ACM International Conference on Software Engineering (ICSE'01), Totonto, Ontario, Canada, 2001.
- [2] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," International Journal of Man-Machine Studies, vol. 18, pp. 543-554, 1983.

- [3] N. Pennington, "Comprehension strategies in programming," presented at Empirical Studies of Programmers: Second Workshop, New Jersey, 1987.
- [4] J. Good, "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension," University of Edinburgh, 1999.
- [5] A. v. Mayrhauser and A. M. Vans, "From program comprehension to tool requirements for an industrial environment," Proceedings of the Second IEEE Workshop on Program Comprehension, Capri, Italy., pp. 78-86, 1993.
- [6] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," presented at Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, 2001.
- [7] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," Commun. ACM, vol. 37, pp. 72-82, 1994.
- [8] C. Simonyi, "Intentional Programming," The Intentional Software Corporation, 2005.
- [9] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," presented at Program Comprehension, 2002. Proceedings. 10th International Workshop on, 2002.
- [10] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*: John Wiley & Sons, 2004.
- [11] T. J. Biggerstaff, "Design recovery for maintenance and reuse," Computer, vol. 22, pp. 36-49, 1989.
- [12] K. Carley and M. Palmquist, "Extracting, Representing and Analyzing Mental Models," Social Forces, vol. 3, pp. 601-636, 1992.
- [13] T. M. Mitchell, *Machine Learning*. New York: McGraw Hill, 1997.
- [14] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," Journal of Software Maintenance: Research and Practice, vol. 7, pp. 49-62, 1995.
- [15] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," Software Engineering, IEEE Transactions on, vol. 29, pp. 210-224, 2003.
- [16] W. Chung, W. Harrison, V. Kruskal, H. Ossher, J. Stanley M. Sutton, and a. P. Tarr, "Working with Implicit Concerns in the Concern Manipulation Environment," presented at Linking Aspect Technology and Evolution Co hosted with Aspect Orientated Software Development (ASOD 05), Chicago, USA, 2005.
- [17] M. P. Robillard, "Representing Concerns in Source Code," The University of British Columbia, 2003.
- [18] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: a project memory for software development," Software Engineering, IEEE Transactions on, vol. 31, pp. 446-465, 2005.
- [19] J. Singer and T. Lethbridge, "Studying work practices to assist tool design in software engineering," presented at Proceedings 6th International Workshop on Program Comprehension IWPC98, Ischia Italy, 1998.
- [20] T. C. Lethbridge, S. E. Sim, and J. Singer, "Software Anthropology: Performing Field Studies in Software Companies," 2004.
- [21] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," presented at Software Engineering, 2003. Proceedings. 25th International Conference on, 2003.
- [22] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," presented at Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004.
- [23] D. Norman, *Things that make us smart : defending human attributes in the age of the machine*: Reading, Mass : Addison-Wesley Pub. Co, 1993.
- [24] J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*: Addison-Wesley., 1984.
- [25] Z. Cornelia and L. Juliane, "Computer-assisted Content Analysis without Dictionary," presented at Sixth International Conference on Logic and Methodology - Recent Developments and Applications in Social Research Methodology, Amsterdam, The Netherlands, 2004.
- [26] J. Diesner and K. Carley, "Using Network Text Analysis to Detect the Organizational Structure of Covert Networks," presented at NAACSOS, 2004.
- [27] R. Popping, *Computer-assisted Text Analysis*. London: Thousand Oaks: Sage Publications, 2000.
- [28] K. M. Carley, "Extracting team mental models through textual analysis.," Journal of Organizational Behavior, vol. 18, pp. 533-558, 1997.
- [29] K. Czarnecki and U. Eisenecker, *Generative Programming - Methods, Tools and Applications*: Addison-Wesley, 2000.
- [30] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," Machine Learning, vol. 29, pp. 131-163, 1997.
- [31] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," Software Engineering, IEEE Transactions on, vol. 28, pp. 970-983, 2002.
- [32] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," presented at Conference on Software Engineering and Knowledge Engineering (SEKE'04), 2004.
- [33] R. C. Prim, "Shortest connection networks and some generalisations," Bell System Technical Journal, pp. 1389-1401, 1957.
- [34] B. Cleary and C. Exton, "The Cognitive Assignment Eclipse Plug-in (ICPC 06)," presented at International Conference on Program Comprehension, Athens, Greece, 2006.
- [35] B. Cleary and C. Exton, "CHIVE - a program source visualisation framework," presented at 12th IEEE International Workshop on Program Comprehension, Bari, Italy, 2004.
- [36] A. LeGear, B. Cleary, J. Buckley, J. J. Collins, and C. Exton, "Making a Reuse Aspectual View Explicit in Existing Software," presented at Linking Aspect Technology and Evolution Co hosted with Aspect Orientated Software Development (ASOD 05), Chicago, USA, 2005.
- [37] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," Journal of Software Maintenance, vol. 11, pp. 201-221, 1999.

A tool-supported environment for knowledge feedback cycle in software development

Noriko Hanakawa

*Faculty of Management Information, Graduate school of Corporation information
Hannan University, Japan
hanakawa@hannan-u.ac.jp*

Abstract

We propose a software development environment including knowledge feedback cycle. A feature of the environment is to support the cycle with tools and researchers. Developers in the environment can acquire knowledge of past projects and experiences without additional efforts of making summaries about past projects. Moreover, the knowledge feedback cycle includes the transfer from parts of tacit knowledge to explicit knowledge. Researcher and tools are assigned to important roles in the knowledge transfer. Therefore, the environment concept will be useful to the semi-automatic knowledge transfer.

1. Introduction

Recently, software development scale becomes bigger, and software quality's impact to our society is significantly increasing. On the other hand, lifetime of software is getting shorter. In order to develop software with certain qualities in a limited time, developers require various knowledge such as cost estimation or risk management, as well as other software engineering techniques and technologies. Some of such knowledge should be extracted and accumulated through their own experiences. However, acquiring and accumulating such knowledge require long time and large efforts. In other words, it is very difficult for developers to become matured engineers in a short period.

In order to help knowledge acquirement and accumulation for novice software engineers, we propose an environment for cycling knowledge among experienced developers, software engineering researchers and novice developers. We call it KFC (Knowledge Feedback Cycle). In the KFC environment, knowledge, mainly concerning risk management and cost estimation, is extracted from past experiences for future reuse. A feature of the environment is semi-automatic. Mainly three tools,

EPM (Empirical Project Monitor), Project Replayer, and Project Simulator, are used to capture and circulate knowledge in KFC. EPM[1] is a tool to automatically collect project data from source code repository, bug-reports and e-mails. Project Replayer is a tool used to review data of past projects. Project Simulator is used to provide actual feedback to developers.

In this proposal, we mainly present the tool-supported environment for KFC. Especially, we show how knowledge is semi-automatically feedback though the environment.

2. A semi-automatic environment for KFC

Developers understand an importance of transfer of past experience for future project. Managers have often experienced bug reports caused by same problems such as insufficient communication with customers. Managers said "Last year, I received similar bug reports caused by insufficient communication". Developers answered "I was not here. I worked at another project last year". This is a typical case in which knowledge and experiences did not transfer to future projects. Because knowledge and experience are shut into personal memories, knowledge and experiences can not be reused as an organization. Of course, many trials of knowledge transfer have been proposed in knowledge engineering field [2]. Tacit knowledge of individuals should be explicit, after that, the explicit knowledge is shared in an organization. However, if people transfer manually from tacit knowledge to explicit knowledge, for example, making rule documents, it will take long time and large efforts. Because of nuisance of making the documents, manual knowledge transfer is problematic. Therefore, we propose a software development environment supporting a semi-automatic transfer of knowledge.

The purpose of KFC is to circulate knowledge and experience of past projects to future projects. Developers are supposed to acquire new knowledge

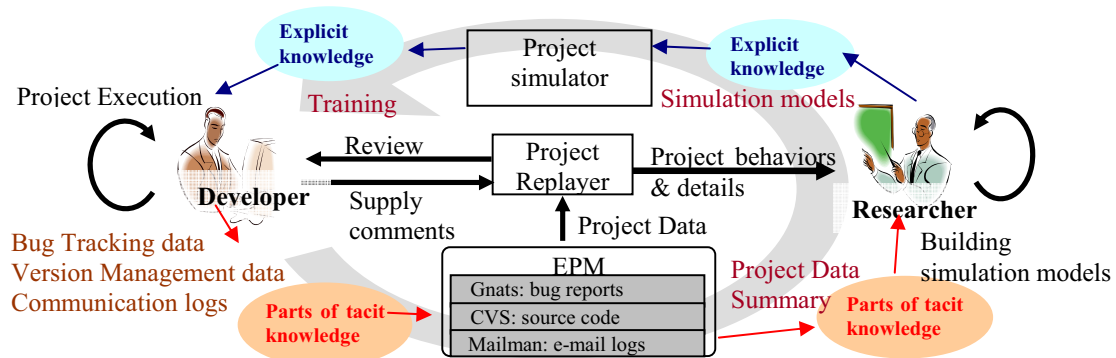


Figure. 1. An environment for Knowledge Feedback Cycle

while experiencing software development projects. To establish automatically the cycle, KFC employs three tools; EPM (Empirical Project Monitor), Project Replayer and Project Simulator (See Figure. 1). A typical scenario in the KFC would be as follows;

- Step1: Various development data (records of code modification, bug tracking, and emails) is automatically captured by EPM during the project enactment.
- Step2: Researchers analyze the collected data to construct various simulation models using the Project Replayer and analysis tools.
- Step3: Using the Project Replayer, developers review past projects. Events and accidents that are not recorded by EPM are also clarified in interview with developers.
- Step4: Regarding results of reviews and interviews, researchers refine their simulation models that were made in Step2. The models are embedded into the Project Simulator.
- Step5: Using the Project Simulator, novice developers learn complicated phenomena in past projects. Developers can also utilize the Project Simulator to make their next project plans. The planned project is regarded as the target of Step1 of the next cycle.

3. A benefits from the environment

It is a most characteristic feature of the environment that the tools and researchers support the knowledge feedback cycle. At first, while developers make usually software, development data (records of code modification, bug tracking, and email logs) is recorded automatically. Knowledge and experiences are implicitly buried under miscellaneous development data. Developers can not identify even their-own knowledge in the development data. After researchers have to analyze the development data, researchers extract knowledge and experiences. In short, researchers transfer from parts of tacit knowledge in the development data to explicit knowledge. The

explicit knowledge is embedded to a simulation model. The formulas of the simulation models present the explicit knowledge. Of course the extraction of knowledge from the development data is difficult. Project Replayer is useful to extract knowledge[3].

Next, by implementing the simulation model to the Project Simulator, the explicit knowledge becomes available to developers. Virtual projects in the Project Simulator behave based on the simulation model. While novice developers are utilizing the Project Simulator, the developers can learn phenomena that occurred in past projects. That is, the novice developers can acquire knowledge of past projects. In KFC, developers do not need any tasks of making the development documents in order to transfer the past knowledge. Developers do only usual development tasks and learning on the Project Simulator. Although researchers have to make simulation models with analyzing the development data, the knowledge transfer is semi-automatic for developers who are true “users” of the environment.

3. Summary

A concept of a semi-automatic environment for knowledge feedback cycle has been proposed. The EPM and the Project Replayer have been already developed. In future, after the Project Simulator will be built, we will confirm the usefulness of the environment.

References

- [1] Masao O., et. al., “Empirical Project Monitor: A System for Managing Software Development Projects in Real Time”, in Proceeding of ISESE2004, Vol.2 , 2004, pp. 37-38.
- [2] Nonaka I, “A Dynamic Theory of Organizational Knowledge Creation”, Organization Science, Vol.5-1, Feb. 1994.
- [3] Goto K., Hanakawa N., Iida H., “Project Replayer-An Investigation Tool to Revisit Process of Past Projects”, Proceeding of International Software Process Workshop and International Workshop on Software Process Simulation and Modeling, May 2006, pp72--pp79.

Social Network Analysis on Communications for Knowledge Collaboration in OSS Communities

Takeshi Kakimoto Yasutaka Kamei Masao Ohira Ken-ichi Matsumoto
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama Ikoma Nara Japan 630-0192
{takesi-k, yasuta-k, masao, matumoto}@is.naist.jp

Abstract

Knowledge collaboration is the key for success of open source software (OSS) communities, because not all members have knowledge and skills necessary for software development. Generally, members in OSS communities communicate for knowledge collaboration using communication tools (e.g. mailing lists, discussion forums, bug tracking systems, and so on) so that geographically distributed members collaborate and coordinate their work. In this paper, we apply social network analysis to the data accumulated in communication tools. We analyzed relationships between the density of social networks and OSS releases by time series analysis of 4 OSS communities in SourceForge.net, in order to investigate the quality of communications for knowledge collaboration. The analysis results showed that communications among community members with a variety of roles are active before/after OSS release in communities where knowledge collaboration is going well.

1. Introduction

Nowadays software developers continuously require a considerable amount of new and diverse knowledge about technologies for software development such as programming languages and components libraries, since such technologies have been evolving from day to day and the past knowledge about them cannot be used soon. In this situation, an individual developer cannot possess every kind of knowledge about latest technologies needed for software development. Knowledge collaboration [11] is not desirable but necessary for modern software development.

Especially, open source software (OSS) development communities rely on knowledge collaboration among community members who have a variety of roles such as com-

munity leaders, developers, bug reporters, passive users and so forth [7, 12], because OSS communities, differently from traditional software development organizations, cannot recruit members who have sufficient skills and knowledge required for building software systems in advance.

In typical OSS communities where community members are geographically distributed, knowledge collaboration takes place through using collaboration tools such as version control systems, bug tracking systems, and mailing lists. Based on the data stored in the collaboration tools, prior studies discussed the model of collaboration processes in distributed environments [10], the efficiency of communication and coordination in distributed software development [4], the benefits of OSS style software development [6], communications metrics for knowing the quality of group work [2] and so forth.

In this paper, we would like to investigate the quality of communications for knowledge collaboration by analyzing the data from communication tools used for distributed software development and the data denoting the success and failure of knowledge collaboration (e.g. number of software releases and number of software downloads). In OSS development, community members rarely meet to discuss but communicate heavily using electronic media (e.g. mailing lists and forums). So, we supposed that we might comprehend the success and failure of knowledge collaboration from the quality of communications among community members through collaborative communication media.

As an approach to inspecting the quality of communications for knowledge collaboration, we use social network analysis (SNA) [8, 9], especially the density of social networks which is a measure to know the quality of social relationships among people (e.g. intimacy or solidarity among people). In this paper, we applied SNA methods to the communication data stored in forums for OSS communities in SourceForge.net (SF.net)¹.

¹SourceForge.net, <http://sourceforge.net/>

In what follows, in Section 2 we hypothesize on communications for knowledge collaboration, more specifically, how knowledge collaboration in OSS communities is conducted using electronic communication media. Section 3 describes density of social networks, which is a measure for SNA. In section 4 we analyze 4 OSS communities in SF.net. Section 5 is the results of our analysis. We discuss the results and our hypothesis in Section 6. Section 7 concludes the paper.

2. Communications for Knowledge Collaboration in OSS Communities

In this section, we discuss communications for knowledge collaboration in OSS communities. Typical OSS communities where community members are geographically distributed and rarely meet to discuss together, heavily relies on collaboration tools such as version control systems and bug tracking systems and electronic communication media such as mailing lists and forums to precede their knowledge collaboration. Yamauchi et al. [10] had conducted two case studies to investigate how OSS development communities achieve smooth coordination and effective collaboration. One of the findings of the case studies was that collaboration and communication tools (e.g. CVS, TODO lists and Mailing lists) were used in a good balance between centralization and spontaneity [10].

In this paper we would like to focus on the quality of communications for knowledge collaboration through communication media. In OSS development, communications for knowledge collaboration involve a variety of people. For instance, software developers discuss technological problems, bug reporters point out bugs of released software, end-users request developers to add new features and so forth. It is important for knowledge collaboration to involve such a variety of community members because “voice” from bug reporters and end users often makes OSS reliable and innovative, and motivates OSS developers to develop further OSS[3].

Figure 1 shows a simple model on a cycle of knowledge collaboration in OSS development. Before OSS released, OSS developers discuss their products and related problems (development period). After OSS released, users ask questions on usage of the products to other users or developers and also report bugs or requests of a new features to developers (feedback period). Again, developers discuss the reported bugs and requested features, and then modify and refine their products. This would be a simple view of a cycle of OSS development but an important aspect of knowledge collaboration, because an end user would not use the products if s/he can get help from other community members, a bug reporter would not report bugs if developers do not modify reported bugs, and developers would not continue

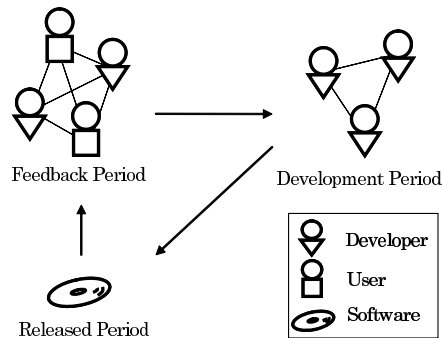


Figure 1. Cycle of Knowledge Collaboration

to create software products if no one use them. Here we can make a hypothesis on communications for knowledge collaboration in OSS development communities as follows.

Hypothesis: Communications are actively encouraged before/after OSS released, especially among community members with a variety of roles but not among particular members.

We thought that we might be able to know the success and failure of knowledge collaboration or “health condition” in OSS communities by analyzing the quality of communications among community members before/after OSS released. The next section describes use of the density of social networks which is our approach to investigating the quality of communications in OSS communities.

3. Density of Social Networks

Using the density of social networks in social network analysis (SNA) is a simple way to know the quality of social relationships among people [8, 9]. Social relationships can be graphed as social networks, which consist of persons (nodes) and their relationships (edges).

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, 9]. The formula for the density of social networks is

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

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.

If social networks show low density, the social relationships tend to be “open” which means *a large, open, diverse, and externally focused relationships* [1]. If social

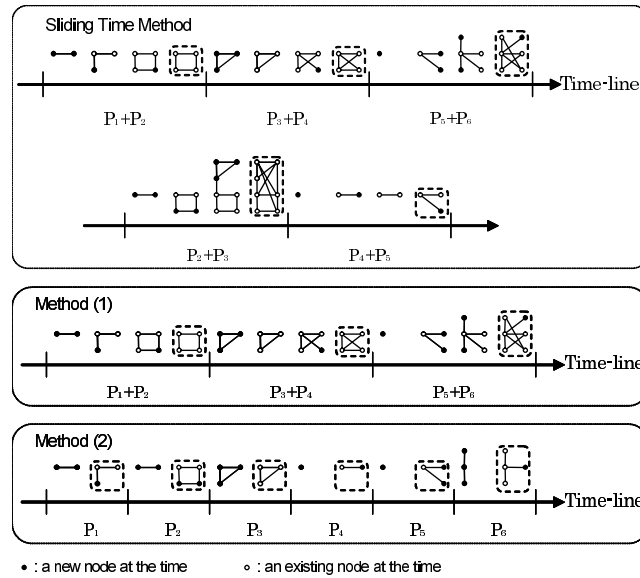


Figure 2. Calculation methods for the density

networks indicate high density, the social relationships often have characteristics of “closed” which means *a small, closed, homogeneous, and internally focused network* [1].

In this paper we apply SNA to the communication data stored in communication tools such as mailing lists and forums (bulletin board systems) to know the quality of communications for knowledge collaboration in OSS development. In this case, social relationships can be defined by posts and replies. Community members (e.g., developers, end-users, bug reporters, and so on) discuss issues related to OSS development. If a member (A) posts a message to a forum for a community (C_i) and a member (B) replies the message, then it can be assumed that there is a social relation between A and B in C_i . Therefore, the density of the social relationships (i.e. social networks) will be high when community members mutually discuss a topic in a forum, but it will be low when no one post a reply message even if there are a number of posted messages in a forum.

Although the activeness of an online community, in general, can be measured by the amount of communications among community members, we do not use the number of posted messages to a forum to know the quality of communications from the above reason. We also do not use the number of replies to know how community members mutually discuss issues because only a handful of members often reply to posted messages in an online forum [5]. We expect that the density of social networks is better to know whether communications for knowledge collaboration are going well or not.

4. Analysis on The Quality of Communications for Knowledge Collaborations

4.1. Dataset

We collected the data involving public forums and OSSs released in 4 OSS communities for the time interval between December 1, 1999 and December 31, 2005. These communities were selected as target communities for analysis because they indicated characteristic measurements results (e.g. a large number of developers, downloads, or posts). We did not collect the data of mailing lists because the mailing lists were not used for communications among community members but mainly for announcements of OSS releases or archives of CVS logs. The data on public forums includes ID of each posted message, user’s name who posted messages, the date of messages posted, ID of each replied message, and ID of each OSS community. The data on released OSS includes the number of developers in each community, the start date of each community, the number of downloads, the number of average downloads per a day, version numbers of released OSS, the release date of OSS, and ID of each OSS community.

4.2. Analysis Procedure

The followings show the procedure of our analysis using social network analysis (SNA) [8, 9].

Preparation Before calculating the density of social networks, firstly we need to define social networks in

Table 1. Characteristics of target communities

Community	Num. of developers	Density of all periods	Num. of posts	Date of communities started	Num. of downloads	Num. of average downloads per a day
Community A	138	0.022	174	04-Jun-01	28,265	16.92
Community B	1	0.013	165	07-Oct-04	7,734,629	17188.06
Community C	11	0.007	766	05-Dec-99	26,000,000	11878.12
Community D	3	0.500	203	29-Dec-03	156	0.21

the context of our analysis. As described before, our aim of using the density of social networks is to know the quality of communications for knowledge collaboration. We use the communication data made from discussions (messages) in forums.

From messages in forums for a target community², we identify who posted a message to the forums (node A) and who replied to the message (node B). Then we regard the relation between the poster (node A) and the respondent (node B) as an edge, by threading relationships between posts and replies as social networks. Repeating this for all messages in forums of a target community, we can graph the relationships as social networks and calculate the density of the social networks.

Calculations of network density by a certain period

Calculating the density of social networks from all the data is inadequate, because the density is calculated from a snapshot of structures of social networks at a certain point while structures of social networks change over time. Therefore, time series analysis is necessary to know changes of the quality of communications among community members, that is, changes of the density of social networks. In order to see temporal changes of the density of social networks, we have to fix a particular time interval.

We calculate the density of social networks from social networks for a period P in a way that slides a $\frac{P}{2}$ interval (sliding time method) in this paper. Figure 2 shows calculation methods for the density of social networks. The density of a social network for a certain period is calculated from the structure of the network at the end of the period.

The sliding time method in this paper is sensitive to changes of network structures than method (1) and (2) which not overlap neighboring periods. For example, communications are active in the period of $P_2 + P_3$.

²A community can have several forums for different purposes of discussions

However, method (1) can not reflect such the activeness. Method (2) which divides the period in half also can not reflect the activeness because it can only show small changes.

In this paper, the density of social networks is calculated by one and a half month ($P = 3$ months). The reason why we fix 3 months is we considered that one topic in a forum is finished about 3 months. We need further consideration for this period or a way to fix an appropriate period.

Time series analysis We analyze relationships between the density of social networks and OSS releases in order to verify our hypothesis. Changes of the density of social networks in time series are used in the analysis. The number of posters who posted messages (i.e. nodes), links among posters (i.e. edges), and posted messages are also used.

4.3. Target Communities

In this paper, we analyze 4 characteristic communities. Table 1 shows the measurement results of each community. In what follows, we describe an overview of each community, which consists of characteristic measurement results, developing software, and usages of forums.

Community A Community A has a number of developers. This community has been developing an operating system for controlling small electronic devices. The posted messages to the forum of community C consist of questions on implementation from developers. This community is currently working on own web site but not on SF.net.

Community B Community B has only one developer but provides a tool downloaded by a large number of users. This community provides windows installers for image manipulation software which is originally developed for UNIX. The posted messages are only from users.

Community C The tool created in community C is downloaded by a large number of users. Community C has been providing a CD ripping tool. The posted messages to the forums of the community consist of posts regarding implementation of software, questions on released software, bug reports, and requests for new features. Both developers and users often post to the forums. Anonymous users who do not have user ID of SF.net also use them.

Community D The characteristic measurement results of community D are that the network density is very high and the number of downloads and posters is very small. Community D creates an OpenGL viewer with command line tools. The forums of this community are used only by developers excepting one post by a user.

5. Analysis Results

Figure 3 shows time series graphs for 4 target communities. The horizontal axis shows time sequence, the vertical axis at the right side is values of the density of social networks, and the vertical axis at the left side is the number of posters, links among posters and posts for each period. Dashed lines mean the date of OSS release.

Community A The following pattern of the changes of the density in community A was repeated. At the initial phase of the community started, values of the density became high. Then, values of the density were decreased as the community progressed. Finally, values of the density became zero. Version 0.6.0 and version 1.0 were released when values of the density became zero. Values of the density before OSS releases were higher than that for OSS release periods excepting version 0.6.1. The number of posts after OSS releases was larger than that for OSS release periods. Posted messages before OSS releases were mainly from developers and posted messages after OSS releases were from users.

Community B In community B, when the density was increasing or high, new versions were released in a short interval. On the other hand, when the density was decreasing, new versions were released in a long interval. Values of the density after OSS releases were higher than that for OSS released periods in most cases. When the number of posts was small in a long interval, community B did not release software products. Developers did not post a message. All messages were posted by users.

Community C In community C, values of the density before OSS releases were higher than that for released

periods in all cases. Values of the density after OSS releases were also higher than that for released periods excepting version 1.50. The degree of increases of density values after releases is decreasing as the community progressed. The number of posts after OSS releases was larger than that before OSS releases. Posted messages before OSS releases were from developers and that after OSS releases were from developers and users.

Community D The number of posters is small against the number of posts in the community D. All messages were posted by developers. In the version 0.1 release, values of the density after the release were higher than that for the released period. No developers posted messages after September 2004.

6. Discussion

The analysis results excepting community D showed that values of the density before OSS releases are high in the community that has a number of posts from developers (community A, C). And, values of the density after OSS releases are high in the community that has a number of posts from users (community B, C).

In the community C that meets both the conditions, values of the density before and after OSS releases are higher than values of the density for released periods. In other words, communications among community members are active before and after released periods in community C. On the other hand, community D that is not the case with these conditions seems to be stagnant as the number of downloads and posters was very small and OSS was released after the version 0.1. Therefore, we consider that our hypotheses are true for communities where knowledge collaboration among community members with a variety of roles is going well.

One of the advantages of using the density of social networks is that we can know community members mutually discuss issues. If the number of posts is large but the density is low, it would mean that many members post messages but do not receive replies from other members. The density may be used for an indicator which reflects a state of knowledge collaboration in their community. So community leaders or managers can help others discuss when the density is very low.

However, we need to note that values of the density are very sensitive against changes of the number of nodes (posters in this paper). Values of the density often show extremely high when the number of nodes is very small. It was very difficult to understand what high values of the density mean in our case study. For instance, the first local peak of the density value (08/25/01) in community C

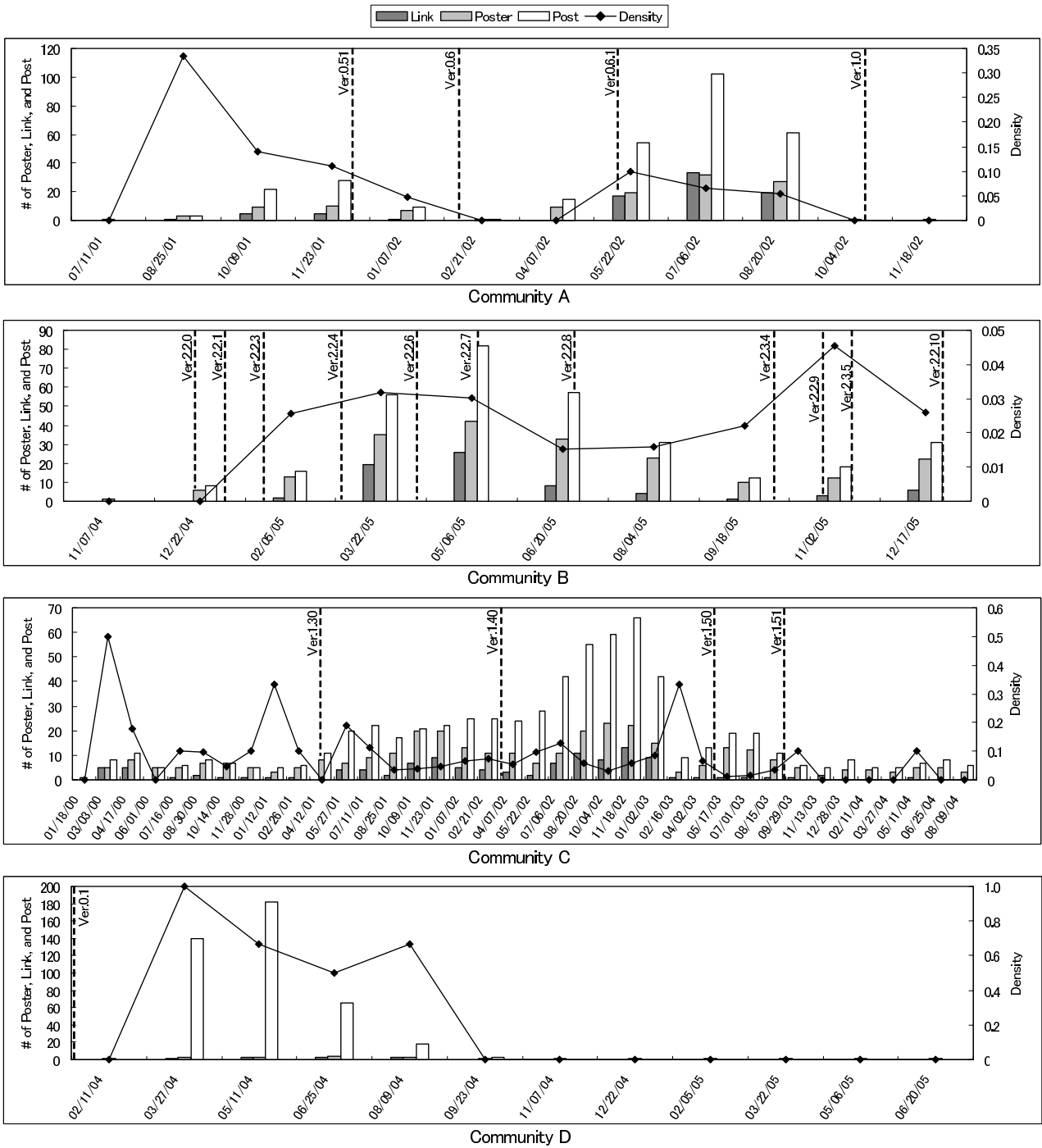


Figure 3. Analysis Results

does not mean knowledge collaboration is going well because only a small number of particular members discuss. This is applicable to community D while community D has a number of posts. In the future, we need to improve this difficulty in using the density.

7. Conclusions and Future Work

In this paper, we investigated the quality of communications for knowledge collaboration by time series analysis using the density of social networks. From the results of analyzing changes of the density in 4 OSS communities, our hypothesis (communications are actively encouraged before/after OSS released, especially among community members with a variety of roles but not among particular members.) was partly verified.

In the future, we will analyze the data by separating developers from end users to distinguish between development periods and feedback periods in more detail. And, weh we also need to analyze the data by considering structures of social relationships among community members though we did not include them in this paper.

Acknowledgments We would like to thank Shinsuke Matsumoto for helping us analyze OSS communities. 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.

References

- [1] W. E. Baker. *Achieving Success Through Social Capital*. John Wiley & Sons Inc., 2000.
- [2] A. H. Dutoit and B. Bruegge. Communication metrics for software development. *IEEE Transactions on Software Engineering (TSE)*, 24(8):615–628, 1998.
- [3] J. Feller and B. Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, 2002.
- [4] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering (TSE)*, 29(6):481–494, June 2003.
- [5] K. R. Lakhani and E. von Hippel. How open source software works: “free” user-to-user assistance. *Research Policy*, 32(6):923–943, June 2003.
- [6] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.
- [7] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSSE’02)*, pages 76–85, New York, NY, USA, 2002. ACM Press.
- [8] J. Scott. *Social Network Analysis: A Handbook*. SAGE Publications, 2000.
- [9] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.
- [10] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: how open-source software succeeds. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work (CSCW’00)*, pages 329–338, New York, NY, USA, 2000. ACM Press.
- [11] Y. Ye. Dimensions and forms of knowledge collaboration in software development. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC’05)*, pages 805–812, Taipei, Taiwan, December 2005. IEEE Computer Society.
- [12] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *Proceedings of the 25th International Conference on Software Engineering (ICSE’03)*, pages 419–429, Washington, DC, USA, 2003. IEEE Computer Society.

The Flow of Knowledge in Free and Open Source Communities

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria
Victoria, Canada
dmgerman@uvic.ca

Abstract

In this paper we present a survey of the methods used by a selection of successful free and open source projects to exchange, store and retrieve knowledge. In particular, we look into mailing lists, Internet Relay Chat, conferences, and code reviews. We explore how historical records left during the development become stored knowledge that can be subsequently retrieved. We also discuss the existence of meta-communities (composed of members of different communities) that allow knowledge to flow from one community to another.

1. Introduction

Free and open source software (FOSS) development has established itself as an effective way to develop software. Perhaps one of its most radical features is that its members are willing to give away knowledge without any direct remuneration¹. This is particularly striking in an era in which intellectual property (mainly in the form of copyright and patents) is highly protected for its economic value. For this reason FOSS has been frequently compared to science where its participants publish their findings into a commons for the benefit of everybody [24].

The “community” is an important concept in FOSS development. It refers to the individuals and organizations that participate in the development of a FOSS application. Some participants are totally passive (by

strictly using an application without ever participating in its development) to totally active (the so called “core” developers who are responsible for most of the contributions to the project. The ways in which a member of the community can participate in the development of a project are extremely wide. Table 1 lists various types of knowledge contributions that individuals can make. Organizations (beside paying developers to contribute to a FOSS projects) can contribute knowledge directly to FOSS projects. For example the original source code of Mozilla was donated by Netscape Corporation; and IBM has pledged to license 500 patents to open source projects [9]).²

One of the main challenges that FOSS projects have is the need to attract and nurture new members. It is, therefore, important to reduce the learning curve of newcomers to ease their integration into the development process and to encourage them to start contributing to the application as easily as possible.

In this paper we are interested in answering the following research questions:

- What are the methods used by a FOSS community to share, store, and diffuse knowledge?
- Is knowledge exchanged between communities, if so, how?

The methodology we have used is based on three main components:

- A literature review regarding how knowledge is created and diffused in FOSS communities.
- A qualitative analysis of the following successful FOSS projects: Apache, Evolution, Linux, GNOME, Mozilla, gcc and postgresQL.

¹ There are many individuals who are paid by a third party to contribute to a FOSS project. In this case we can consider these individuals as “proxies” of the organization who hire them. These organizations are, therefore, contributing knowledge without expecting a direct remuneration for their contributions—but there are most likely seeking indirect benefits.

² Organizations can also indirectly contribute knowledge in the form of training and diffusion, and by paying employees to become contributors.

Type of contribution	Description
Source code	This is perhaps the most visible contribution.
Documentation	In the form of Web sites, user and developer manuals, magazine and Web articles, books, FAQs, etc.
Internationalization	Translations of the software and documentation into different languages.
Code Reviews	The discussion and improvement of source code contributions.
Testing and debugging	Formal or informal testing and debugging.
Bug reports	Submit bug reports that can be used by the development team to track and fix defects.
Configuration management and build process	Tasks required to maintain the environment necessary for multiple developers to participate.
Distribution of binaries	Preparation of binaries for download by any user interested to try the software.
Suggestions	Ideas on how to improve the product.
Answers to developer's questions	They help other developers who are contributing.
Answers to user's questions	They help individuals who are trying to use the software.
Release management	Dedicated to prepare and advertise new releases.
Legal	They provide information regarding legal issues, such as licensing, and other intellectual property issues.
Web site development and maintenance	These contributions usually gather knowledge from other sources and make sure it is persistent. It can also include those who contribute to wikis.
"Pointers" to knowledge	Perhaps the smallest type of contribution it involves answering a question by "pointing" to another source of information (such as a Web site or a research article).
Distribution packaging	Knowledge needed to prepare packages to be included in distributions (such as SUSE, Red Hat, Fedora, etc).

Table 1. Type of Knowledge Contributions to a FOSS project

- The experiences of the author as a contributor to several FOSS projects ³.

The paper is divided as follows. Section 2 addresses the question of how knowledge is shared, stored and diffused within a FOSS community. Section 4 analyzes how knowledge is exchange across FOSS communities. We conclude with a discussion of our findings and directions for future research.

2. How does knowledge flow within a FOSS community?

Research has shown that two important motivations that individuals have to become FOSS software developers are to: 1) improve their career perspectives (by acquiring and refining skills) and 2) be recognized in the meritocracy of a FOSS community[14, 8]. This implies the existence of knowledge flow in FOSS communities from those who have it to those who seek

³ The author is currently one of the core developers of Panotools. Panotools is a group of tools to combine two or more photographs into a panoramic one, see `panotools.sourceforge.net`.

it. Given the variety of knowledge required to produce software (programming skills, application domain, management skills, marketing skills, etc) individuals might become both a producer and a consumer of knowledge depending on the skills that they bring to a given project (and the skills that they are particularly interested in learning and improving).

The flow of knowledge from one individual to another requires the creation and development of an infrastructure that supports it. It is also necessary to create mechanisms that permit its short and long term storage and retrieval.

From its beginning the Internet and FOSS have co-existed in a symbiotic manner: the Internet was born thanks to the sharing of source code and source code has thrived as the Internet matures. It is undeniable that the Internet is the main channel over which knowledge flows within a FOSS community. Project-sponsored conferences (see section 2.3) are perhaps the only form of exchange of knowledge in FOSS that does not require the Internet (although it uses it for its organization).

As FOSS projects evolve their communities evolve as well: new members join, some leave. At the same

time some of its members shift their roles, depending on many factors, including the time they can invest to the project [18]. Nakakoji et. al use *Legitimate Peripheral Participation* theory (LPP) to explain this evolution: “a community of professionals evolves by reproducing itself when peripheral new members (i.e. apprentices) become fully qualified members (i.e. masters). The process of becoming a master is the process of learning. [...] the community member acquires the skills and knowledge embodied in the community by interacting with master members” [18]. One important conclusion of the study by Nakakoji et. al is that the evolution of FOSS communities is determined by two factors: “the existence of motivated members who aspire to play roles with large influence, and the social mechanism of the community that encourages and enables such individual role changes” [18].

Like in any other software development team, members of a FOSS community eventually leave it for multiple reasons (these can range from lack of motivation or available time, to passing away). Without a constant influx of new members, any FOSS community will eventually collapse. A FOSS community, therefore, requires the flow of knowledge from one member to another, and the storage (temporary and permanent) of that knowledge so it can be retrieved and reused by new members (this knowledge becomes the project’s community memory).

2.1. Email

Email is ubiquitous as a medium for the flow of knowledge in FOSS. A project usually starts with a mailing list that links developers and users (active—those that contribute to the discussions—and passive—those that only use the product without contributing anything in return). In an empirical examination of 100 FOSS projects Krishnamurthy found that most of them have very few contributors and, on the average, have 2 mailing lists [13]. As a FOSS community grows its discussions are split into different types of mailing lists.

The most commonly found lists are those dedicated to *announcements*, *users*, and *developers*. In the large projects that we analyzed we found that mature, widely used projects tend to have highly specialized mailing lists (Mozilla, for example, has 81). We discovered that there exist five main types of mailing lists:

- **Announcements.** Typically a low traffic, moderated list, it is intended to be used for announcements regarding the status and evolution of the project.

- **Users support.** Mailing lists dedicated to help members who have questions regarding how to use the product.
- **Development.** Developers use them to discuss the development of the project. Some project tend to have very specialized development lists (for example, Apache has a `packagers-` list for the discussion of issues related to how apache is packaged, distributed and made available to users).
- **Software process related.** The messages in these lists are usually produced by tools that support the development process (for example, a version control mailing list that has one message per source code commit, or a bug mailing list that has one message per bug reported).
- **Documentation.** These lists are dedicated to the discussion of documentation and the Web sites of a project.

We also found that all the surveyed projects archive their mailing lists and the majority provide some type of searching mechanism to them.

2.2. IRC

IRC (Internet Relay Chat) is an old Internet protocol that supports many-to-many instant communication. IRC has been widely used to link communities even before the advent of the World-Wide Web. Although it is rarely reported, many FOSS projects have IRC channels where different contributors can meet and exchange knowledge. Apache, for example, uses the IRC channel `#Apache` in `irc.freenode.net`⁴. The flow of knowledge in the Apache IRC channel is demonstrated by Rich Bowen (who is a member of the Apache Foundation and contributes documentation to the server). He writes a monthly column based on his experiences in the Apache’s IRC channel[1].

GNOME has its own IRC server that hosts more than three dozen channels⁵. Similarly Mozilla maintains its own IRC server with more than two dozen channels (almost 60% of them are in languages different from English)⁶.

IRC channels provide a very informal place to exchange information at all different levels. Perhaps its main drawback is that its discussions are rarely

4 `irc.freenode.net` holds several hundred IRC channels for FOSS projects, including postgresSQL, the Free Software Foundation, RedHat, and mySQL <http://freenode.net/primary-groups.shtml>.

5 <http://gnomesupport.org/wiki/index.php/IrcChannels>

6 <http://irc.mozilla.org/>

archived. They are similar to informal verbal conversations happening in the offices and halls of an organization.

2.3. Conferences

Several FOSS projects are organizing conferences where developers can meet face to face. Conferences are usually organized around presentations that are intended to exchange knowledge, or to train other contributors. They are also a place where discussions regarding the future of the project usually take place. From the projects that we surveyed these organize conferences:

- **GUADEC.** This is the GNOME Developers conference (it has taken place once every year since 2000)⁷.
- **ApacheCon.** The Apache Conference, like GUADEC, has run every year since 2000. This year it has three versions: Europe, Asia and US⁸. GUADEC and ApacheCon are two of the oldest running conferences organized by a FOSS community.
- **PostgreSQL Anniversary Summit.** The 10 year anniversary of PostgreSQL is being marked with the project's first conference.⁹

2.4. Code reviews

Code reviews or code inspections were introduced by Fagan as a formal process in which the development team invests time and energy to review the code being produced [3]. In the most formal approach, code reviews are conducted during meetings for which the developers are expected to prepare. These meetings can result in the detection of defects or in recommendations on how the source code can be improved.

Because FOSS developers are usually geographically dispersed they are unable to conduct formal code reviews. Instead they conduct asynchronous reviews using email as the main communication channel. In an empirical study of software inspections Johnson and Tjahjono found no significant differences between meeting-based, and asynchronous code reviews. They did, however, found that the total effort required in meeting-based reviews was significantly higher when compared to asynchronous reviews (and hence the effort to find a bug was higher in meeting-based reviews) [10]. In FOSS code reviews have two main objectives:

- They minimize defects and provide better, cleaner code with less total effort.
- They improve the skills and knowledge of the reviewers and authors of the code.

Few FOSS projects have a formal process for code reviews, and those that do are usually mature, and expected to be reliable. From the projects we studied only Apache, Mozilla and Linux include code reviews as part of their development process. Mockus et. al found that Apache had a similar defect density than several commercial products. They argued that “fewer defects are injected into the [Apache’s] code, or that other defect-finding activities such as inspections are conducted more frequently or more effectively.” [17].

In Mozilla every contribution should be reviewed by at least two independent reviewers. The first type of review is conducted by the module owner or the module owner’s peer (every module has an owner and a set of peers—individuals who are knowledgeable on that module). This review catches domain-specific problems. A patch that changes code in more than one module must receive a review from each module. The second type of review is called a super review. The goal of the super review is to find integration and infrastructural problems that may effect other modules or the user interface¹⁰. By requiring both types of reviews Mozilla ensures that someone with domain expertise and someone else with overall module and interface knowledge have approved the patch.

The Mozilla maintainers acknowledge that super reviews are a good way for “intermediary and advanced training”, but “are a terrible mechanism for training in basic practices”. The main concerns Mozilla maintainers have is that super-reviewers are very few and do not have the time to train other contributors: “[a] super-review [should] be the last stop for training.” [23].

Over the years Apache has experimented with different types of code reviews. Apache currently uses a Commit-Review model, where core developers are allowed to commit changes that are then expected to be reviewed. The review takes place in the developers mailing list, an open environment where any contributor can participate. In an empirical study of code reviews on Apache we found that 9% of post-reviewed commits generated a discussion [21]. Apache post-commit reviews are not only useful as a way to find and eliminate defects, but because they happen in

⁷ <http://guadec.org/>

⁸ <http://apachecon.com>

⁹ <http://conference.postgresql.org/>

¹⁰ Mozilla’s core review process requires the identification of individuals as module owners, module peers, and super reviewers. We can consider these as “knowledge” roles. Research is needed to understand how are these roles filled and who fills them.

a public forum (a mailing list) they also create awareness and diffuse knowledge, even to those who are not active participants in the review.

Code reviews practices in FOSS have started to influence industry. In [15] Lussier described how his company development process was (unexpectedly) influenced by their experiences participating in an FOSS community. Lussier was surprised that the code review practices of the Wine project resulted in better code, always ready to be released. His company decided to introduce a similar process in-house.

3. Storing and Retrieving Knowledge

As a software project evolves, a wealth of information is created (some automatically, some manually). Some of this information records communications between its contributors and users; other explains how the software system is evolving. We have previously demonstrated that historical records can be used to successfully reconstruct how a software system evolves [5].

In our research into the evolution of FOSS projects we have found that developers of mature FOSS projects value these records and ensure, often through policy, that these records be maintained; they form what Cubranic calls the “community memory” [2] of the project. In a study of historical records kept by FOSS communities we have observed the following types [7]:

- The source code itself. Version control systems allow developers to inspect the state of a file at any given time in the past, helping them understand how the system evolves. Source code sometimes is used as a communication medium, where notes and TODO lists are embedded as source code comments (such as the ones described in [25]).
- Defect tracking databases, such as Bugzilla, are frequently found in large FOSS projects. They provide a valuable source of information regarding defects (and their fixes) and feature requests.
- ChangeLogs are files that are usually updated when the system is changed, and provide a description of the given change. The Free Software Foundation requires all its projects to have a ChangeLog file. In those projects that have them, we have discovered that they are almost always properly updated [6].
- The Version Control logs of mature projects tend to have large, meaningful explanations. In the project Evolution, the average size of a log is 306 bytes, in Apache 1.3 it is 160 bytes, and in PostgreSQL it is 160 bytes, to cite just a few.

- Email is seen as an important source of discussion about the way software evolves.
- Code reviews are valuable discussions that provide good insight on why certain changes are performed the way they do. [21]. In contrast, version control logs and comments are shorter, usually omitting discussion of less satisfactory solutions. Having a link to a discussion might save the maintainer many hours in code comprehension and avoids time wasted trying to figure out why a given part of the system was implemented in a certain way.
- Documentation, including Web sites and wikis. FOSS projects are frequently using version control systems to store this type of information, which will allow contributors to inspect their state at any given date.

Some sources of information have a well defined format, such as version control logs and ChangeLogs, and are easy to correlate to lines of affected code. Correlating Bugzilla and source code is more difficult. It usually involves textual analysis of the description of the version control log. For example in [6], we describe regular expressions that were useful in the extraction of Bugzilla numbers from CVS commit logs. Correlating email messages is even more difficult. For Apache, we have been successful in creating automated and manual heuristics that help in the correlation of messages discussing code reviews [21]. Code reviews often involve *diffs* that contain the version in the repository against which the diff was made. However, general email discussions are much more difficult to correlate. Problems include determining the context of the discussion, reconstructing message threads, and resolving names to email addresses.

In [7] we proposed the concept of Evolutionary Annotations (EA), documentation that describes how a software system is evolving. EAs are information extracted (some automatically, some manually) from historical software development records. The purpose of evolutionary annotations is to explain why a project evolves in the way it does (contrary to documentation, that explains what the “current” system is doing). We proposed methods to retrieve them and correlate them to the source code, and described the design and implementation of a prototype for Eclipse that can filter and present these annotations alongside their corresponding source code.

3.1. How are communities using historical records?

Without controlled experiments it is difficult to determine how contributors use the historical information of a project, mainly because it is difficult to identify when a contributor access historical records, and for what purposes.

We have found, however, evidence that the information is being used by developers. Figure 1 shows excerpts from 2 email messages in which an question is answered by providing a link to older email discussions. One particular service that appears to be useful to contributors of FOSS is `Gmane.org`. `Gmane` is dedicated to provide three main services to FOSS mailing lists: archiving of messages (including permalinks), presenting email lists with a Web interface (including a blog-like option, and RSS feeds), and a powerful search engine. As the examples in figure 1 show the service provided by `Gmane` is being used by FOSS communities to retrieve and reuse the knowledge stored in their mailing lists.

The existence of tools intended to extract information from version control logs (such as `Bonsai`¹¹, `cvschangelog`¹², `CVS History`¹³, `CvsGraph`¹⁴, `ViewVC`¹⁵ and many others) suggests that version control logs are useful to the developers. Unfortunately there have been no studies that try to understand how contributors (in both FOSS or proprietary systems) extract knowledge from version control repositories, in which circumstances it is useful, and how this extraction can be improved.

4. Is knowledge exchanged between communities?

One of the greatest assets that a FOSS project has is the size and diversity of its community.

In the proprietary software world knowledge is exchanged between organizations in very few ways. For example, a organization hires an employee from another organization, or by creating “knowledge exchange” contracts where an organization is willing to exchange its knowledge with another in exchange for some consideration. FOSS exchange of ideas and knowledge is often compared to that of science, where knowledge is created and exchanged without any requirement for compensation¹⁶ [24].

11 <https://www.mozilla.org/bonsai.html>

12 <http://cvschangelog.sourceforge.net>

13 <http://cvshist.sf.net/>

14 <http://www.akhphd.au.dk/~bertho/cvsgraph/>

15 <http://www.viewvc.org/>

Most FOSS communities have as their main goal the creation of a FOSS product, and the exchange and flow of knowledge and information is a side effect of it. While it is true that most FOSS projects have very small communities (one main contributor, with very few users), some communities have been able to achieve large numbers. The larger the community, the larger the pool of knowledge available to it. Even though most contributions come from few developers, any given knowledge contribution can have an important impact on the project. These contributions take many different forms: for example, pointers to sources of information (a person posts to a mailing lists a URL to knowledge stored by another project) or domain knowledge (in many cases users are more knowledgeable about the domain of an application than the core developers). Even just a note saying: “this program works great under ‘such’ operating system” might provide valuable knowledge.

FOSS projects are usually part of a larger FOSS ecology. They depend on other applications, and other applications might depend on them¹⁷. This creates a meta-community, where contributors and users from one community contribute (directly or indirectly) to the other communities. It is not uncommon for contributors of one project to subscribe to mailing lists in another project to gain awareness of where the project is and how it is evolving. In [22] Spinellis and Szypersky described how the Xine multimedia player¹⁸ required 11 different libraries. Xine developers, therefore, required to know what these libraries did, and changes in these libraries would have had an effect on Xine, making them stakeholders (and users) in their development. Madey et al [16] used network analysis to demonstrate that FOSS projects create large clusters (a project is related to another project if they share at least one common contributor). They found that the largest cluster in Sourceforge connects 35% of its projects. Research is needed to find out if and how knowledge flows from one community to another via its common contributors.

16 In recent years it is more frequent to find researchers who are opting for patenting their ideas before they make them public.

17 In some cases commercial applications are part of this ecology. `panotools` is a library and collection of applications that are used by some commercial applications (PTGui and PTAssembler). These applications are very interested in fixing bugs and improving `panotools` and have contributed to its development; furthermore, the users of those commercial applications are indirect users of `panotools`—which benefits from their bug reports and suggestions.

18 <http://xinehq.de/>

```
[..]
> We switched physical mail servers and in transferring our ezmlm
> mailing lists and the vpopmail/qmailadmin installation ran into some
> problems. First, all mailing lists are freezing when receiving an
> e-mail from a subscriber with the following message in qmail-send
> log: "Sorry,_substitution_of_
> target_addresses_into_headers_with_#A#>_or_#T#>_is_unsafe_and_not_permitted./"
```

This thread may help you :
<http://article.gmane.org/gmane.mail.ezmlm/4297>
<http://article.gmane.org/gmane.mail.ezmlm/4298>
[..]

```
[..]
> I have seen several posts for this but none resolve my issue.
```

You haven't been looking hard enough ;-)
<http://www.lowagie.com/iText/faq.html#jsp>
<http://article.gmane.org/gmane.comp.java.lib.itext.general/8850>
[..]

Figure 1. Excerpts from mail messages that reference older discussions.

4.1. The Slashdot Effect

Blogs have an influence in the exchange of knowledge among the FOSS communities. Slashdot (slashdot.org, “News for Nerds, Stuff that Matters”) is a blog dedicated to the discussion of technology news, particularly those of interest to FOSS communities [19]. News entries are usually submitted by readers. Its infrastructure enables readers to post comments to the entries and to rank those submitted by their peers (in an effort to improve the signal-to-noise ratio of comments). A special section called “Ask Slashdot” invites readers to submit questions that might be of general interest, expecting other readers to post answers to the questions. Slashdot conducts interviews and publishes book reviews too.

Slashdot provides a place where members of different FOSS communities can gather and discuss issues that might be of their interest. Slashdot is particularly useful to provide awareness, e.g. what other FOSS communities are doing, and issues that affect FOSS in general. It also serves as a place to advertise important advancements in a FOSS project.

Another site worth mentioning is Groklaw¹⁹. Groklaw specializes in the discussion of intellectual property law and its effects on the FOSS world. It was formed in 2003 as a response to the legal chal-

lenges brought by SCO against IBM and other organizations with regard to some intellectual property found in the Linux Kernel (for an overview of the legal case see [4]). Groklaw’s model is very similar to Slashdot’s, although it maintains stronger editorial control, resulting in higher quality of entries and comments. Groklaw demonstrates that a Web site can link two different communities (in this case typical FOSS contributors and legal experts) to create an environment where knowledge is shared, exchanged and enhanced between them. This is encapsulated in a comment by its creator, Pamela Jones: “Some of the volunteers knew things I didn’t, especially about the code issues, but they didn’t realize what they knew was useful legally. [...] People are hungry to understand legal news, and they want to help.” [11].

5. Discussion and Future Work

This paper reports preliminary results on how knowledge flows in FOSS communities. We have found that FOSS communities have developed multiple methods to communicate and exchange knowledge. None of the projects surveyed has exactly the same methods. This can be due to one or several factors: for example, their application domains are different; or their communities work better with different methods.

¹⁹ <http://groklaw.net>

We require a lot of research in this area. We need to perform quantitative empirical studies on how FOSS projects generate, share, store and retrieve knowledge. We also need to perform controlled experiments to compare methods and to understand their advantages and disadvantages, and under which scenarios they can be useful. The results from these studies can help FOSS communities to select the methods better suited for their particular needs. We need to explore new methods to exchange and store knowledge, and equally important, how to make it easier to find knowledge (either who has it, or by finding and retrieving it).

Acknowledgments

The author would like to thank the anonymous reviewers for their suggestions to improve this paper. This research is supported by the National Sciences and Engineering Research Council of Canada.

References

- [1] R. Bowen. A day in the life of #apache. O'Reilly ON-Lamp.com Apache DevCenter, 2003-2005. Montly column.
- [2] D. Cubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: A case study for software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 82–91, 2004.
- [3] M. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.
- [4] L. Geppert. Battle of the Xs. *Spectrum*, 40(8):16–17, Aug. 2003.
- [5] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.
- [6] D. M. German. An empirical study of fine-grained software modifications. *Journal of Empirical Software Engineering*, 2005. Accepted for publication Sept 25, 2005, to appear in the Special Issue of Best Papers of ICSM 2004.
- [7] D. M. German, P. Rigby, and M. A. Storey. Using Evolutionary Annotations from Change Logs to enhance Program Comprehension. In *3rd International Workshop on Mining Software Repositories (MSR 2006)*, May 2005.
- [8] G. Hertel, S. Niedner, and S. Hermann. Motivation of software developers in open source projects: An Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32:1159–1177, 2003.
- [9] IBM Corporation. IBM Statement of Non-Assertion of Named Patents Against OSS. <http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf>, Jan. 2001.
- [10] P. M. Johnson and D. Tjahjono. Does every inspection really need a meeting? *Journal of Empirical Software Engineering*, 5(3):9–35, 1998.
- [11] P. Jones. EOF: open legal research. *Linux J.*, 2004(121):13, 2004.
- [12] B. Kogut and A. Metiu. Open-source software development and distributed innovation. *Oxford Review of Economic Policy*, 17(2):258–264, 2001.
- [13] S. Krishnamurthy. Cave or Community? An Empirical Examination of 100 Mature Open Source Projects. *First Monday*, 7(6), June 2002.
- [14] K. R. Lakhani and B. Wolf. *Perspectives on Free and Open Source Software*, chapter Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects, pages 3–21. MIT Press, 2005.
- [15] S. Lussier. New tricks: How open source changed the way my team works. *IEEE Software*, 21(1):68–72, 2004.
- [16] G. Madey, V. Freeh, and R. Tynan. *Free/Open Source Software Development*, chapter Modeling the F/OSS Community: A Quantitative Investigation, in *Free/Open Source Software Development*. Idea Publishing, 2004.
- [17] 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.
- [18] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, New York, NY, USA, 2002. ACM Press.
- [19] N. Poor. Mechanisms of an online public sphere: The website Slashdot. *Journal of Computer-Mediated Communication*, 10(2), 2005.
- [20] E. Raymond. *The Cathedral & the Bazaar*. O'Reilly, 1999.
- [21] P. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS -305-IR, University of Victoria, 2006.
- [22] D. Spinellis and C. Szypersky. How is Open Source Software Affecting Software Development. *IEEE Software*, 21(1):28–33, Jan-Feb 2004.
- [23] The Mozilla Foundation. Frequently Asked Questions about mozilla.org's Code Review Process. <http://www.mozilla.org/hacking/code-review-faq.html>, June 2006.
- [24] J. Willinsky. Unacknowledged convergence of open source, open access, and open science. *First Monday*, 10(8), August 2005.
- [25] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *MSR '05: Proceedings of the 2005 International Workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

Using SNS Systems to Support Knowledge Collaboration

Masahiko Ishikawa
Software Research Associates, Inc.
masahiko@sra.co.jp

Abstract

SNS (Social Networking Service/Social Networking Site) is known as a tool for promoting social relationships, and has recently attracted a lot of attentions from both academic researchers and industrial practitioners. This paper describes the current status of SNS, and delineates the requirements and action items for applying SNS as a tool to support knowledge collaboration.

1. Introduction

According to a report by the Japanese Ministry of Internal Affairs and Communications, the total number of users who has registered in one of SNS systems has reached about 3,990,000 in 2005[1]. The mixi site, a major SNS site in Japan, has more than 2,000,000 registered users by Dec 2005. In addition to homegrown domestic SNS sites, many SNS providers from abroad also have established operations in Japan. Orkut is one of the most notable examples. It was reported that the Cyworld, a major SNS provider from Korea has started its business in Japan since Dec. 2005.

Driven by the rapid development of SNS systems, research efforts for analyzing the SNS systems and users have also stepped up. For example, Takai and Kawaguchi have conduct experiments with the ACS system that they have developed and evaluated the system's impacts on various human relationships [2][3].

2. Problems of general SNS systems

For new members to join, most SNS systems require invitations by friends who have already participated in the system. Consequentially, SNS users tend to think it is safe and comfortable because many of their friends have joined and they are invited by their trusted friends, and are willing to collaborate with other users.

Although SNS systems are good at managing and promoting human interaction and communications, a general SNS system is insufficient to support

knowledge collaboration. As a user of and developer for SNS systems, I found few cases that meaningful output has been produced by collaboration among members in SNS systems.

It is especially hard for SNS users to engage in knowledge collaboration because of the lack of following support in SNS systems:

1. Personal space area cannot be easily used as a repository of individual knowledge.
2. There is no easy way to trace the discussions among members.
3. It is difficult to store an individual's knowledge in a group space.

3. Motivating people to contribute

Knowledge collaboration makes progress through providing and receiving knowledge among knowledge workers. Ideally, each knowledge worker should be able to act evenly as both a knowledge provider and a knowledge receiver. However, in reality, only a few knowledge workers are providing knowledge and most users act only as knowledge receivers. This fact points to the need of incorporating into SNS systems mechanisms of motivating more members to become active knowledge providers. For example, if an SNS system has the functionality of evaluating contributions made by its members and of ranking SNS users according to their contributions, SNS system might be able to become a more apt environment for knowledge collaboration among its members.

4. Integration with traditional software tools

In the field of software development, software developers have been using many collaboration tools, such as:

1. Bug tracking systems: Bugman, Gnats, Kagemai
2. Full text search engines: Namazu
3. Configuration management tools: CVS, Subversion.

However, my survey of existing SNS systems has found that no systems have tried to learn from those collaboration tools or attempted to integrate with such tools.

5. Research agenda

There is still a long way for utilizing SNS system to support knowledge collaboration. In this section, I try to identify a few action items that need to be addressed.

5.1. Resource synchronization between SNS systems and local computers

Nowadays, many user have a powerful PC with rich resources (large amount of memory, disk storage, gigabit Ethernet and powerful CPU). They often make their documents on their local computers and then upload them into SNS sites. Once they upload the files, the same file exists both in the SNS system and in the local computer. Once they made modification to the files in one place, those files tend to become different. Most current SNS systems do not support synchronization of user files.

5.2. Utilizing cell-phones as input devices

As cell phones become more ubiquitous and move increasingly toward the role of personal digital assistant, many users use such portable devices as a place to write notes and to keep their ideas, and then upload them into SNS sites. An easy transfer of documents between portable devices and SNS systems is needed.

5.3. Connecting to other systems

Some users prefer to use IRC systems like MS-chat, and if they happen to have generated interesting ideas during the chat, they cannot easily make connection into SNS systems and share their newly gained insights with others.

5.4. Inter-SNS collaboration

No SNS systems support inter-site connection with other SNS systems. When different organizations use different SNS systems, inter-organization collaboration then becomes impossible.

6. Summary

In this position paper, I try to summarize the problems to use SNS systems to support effective knowledge collaboration from the perspectives of both a user and a developer.

8. References

- [1] Ministry of Internal Affairs and Communications, "Number of Registered Users of Blogs and SNSs" http://www.soumu.go.jp/s-news/2005/051019_2.html
- [2] K. Takai and N. Kawaguchi, "ACS: A Social Networking System for Various Human Relations" The 20th Annual Conference of JSAI, 2006
- [3] K. Takai and N. Kawaguchi, "The Specification and Construction of the Academic Community System."

Building the Knowledge Network in Software Project

Atsushi Inuzuka

*Japan Advanced Institute of Science and Technology, Japan.
ainuzuka@jaist.ac.jp*

Abstract

One of the most important coordination techniques for software development is to build the effective knowledge network in a software project. The knowledge network, in this paper, refers to inter-functional relationships for obtaining customer needs. We investigated the knowledge networks employing a survey instrument to collect data from a variety of product processes in a Japanese SI (Systems Integration) firm. Our results indicate that we must take a contingency view into consideration to build an effective knowledge network in a software project.

1. Introduction

Information systems theory literature stress the importance of coordination, which refers to “the integration or linking together of different parts of an organization to accomplish a collective set of tasks” [1]. Since software development is a highly information-intensive work activity, a successful software requires tight coordination among the various efforts involved in the software development cycle [2]. However, the main concern of studies on project coordination so far have been the mechanisms or actions taken in projects (e.g., decentralization, formalization), not the actual interaction within projects.

In this paper, we tried to identify inter-functional relationships for obtaining customer needs in software development, what we call a knowledge network. We used the term “knowledge” because of that customer needs do not fit a specific mold. The interpretation of it is highly subjective and socially constructed and has much tacit dimension. For this reason, the term “knowledge network” may fit more to represent the personal interaction than the term “information network.”

The structure of inter-functional relationships has had much attention in management studies. However, the knowledge network we report here is different from these studies in two respects. First, we are concerned only with the interactions of obtaining customer (client) needs. Today, identifying client requirements is critical to the success of a software project, especially for which offers solutions for their customer. It is apparent that personal interactions are critical for a success, however, prior studies have discussed the extent of the interaction among functions and then, what content of information is actually exchanged is not apparent. Limiting the content of interaction as customer needs will help clarify the effectiveness or efficiency of relationships.

Second, we took a contingency view in this problem by investigating which different knowledge networks are actually used and how they affect the success or failure of obtaining customer needs under specific conditions. The information-processing model introduces the concept of organizational information processing as an explanation for why context and structure should match for optimum organizational performance [3][4]. The consensus is that organizational performance is accomplished by the match or fit between the amount of information needed and the organizational information-processing capacity. However, taking the tacit dimension in customer needs into consideration, attention should be focused on knowledge processing not just information processing. Then, proving into the knowledge network under each condition can help to establish ideas for designing configurations that produce optimal performance.

The rest of the article is organized as follows. In the next part, we present a theoretical background for this problem. Subsequently, we empirically show the knowledge network and levels of obtaining or reflecting customer needs by a survey. Comparing the results under each environment, we then show the necessity of a contingency view for building a knowledge network within a firm.

2. Background

2.1. Coordination Mechanisms for Software Development

Coordination mechanisms in software projects have been the focus of a number of investigations. Researchers identified several specific coordination mechanisms, including standards, hierarchies, targets or plans, slack resources, vertical information systems, direct contact, liaison roles, task forces, and integrating goals. Sabherwal classified these mechanisms into four main categories, as shown in Table 1 [5].

Table 1. Categories of Coordination Mechanism

Coordination Category	Examples
Standard	Compatibility standards Data dictionaries Design rules Error tracking procedures Modification request procedures
Plans	Delivery schedules Project milestones Requirements specifications Sign-offs Test plans
Formal Mutual Adjustment	Code inspections Coordination committees Design review meetings Hierarchies Liaison roles Reporting requirements Status review meetings
Informal Mutual Adjustment	Co-location Impromptu communication Informal meetings Joint development Transition teams

Though many types of coordination exist, the importance of personal interaction is unshaken. Kraut and Streeter empirically investigated under what conditions various coordination techniques for software development work well and concluded that personal communication was the critical factor for success [2]. The importance of personal communication would be more apparent when taking our concern, obtaining customer needs, into consideration. That is because customer needs have much tacit dimension in itself, and then, sharing of it is expected to require much personal interaction. Additionally, since software development is a highly social and interactive process, project coordination strategies must exhibit communication mechanisms

that match or fit the task and social context associated with specific work units and project phases [6]. Therefore, characteristics (e.g., structure or density) of a knowledge network could vary in accordance with organizational environments such as customer type, task characteristics, or management constraints faced by organizations.

2.2. Contingency Factors for Coordination

What types of interactions are appropriate under what conditions is the primary concern for contingency theorists. The term contingency theory was coined by Lawrence and Lorsch [7], who argued that the amount of uncertainty and rate of change in an environment impacts the development of internal features in organizations. To cope with these various environments, organizations must create specialized sub-units with differing structural features: e.g., differing levels of formalization, centralized vs decentralized, planning time horizon [8]. Taking the contingency view into software development, appropriate inter-functional interaction (coordination) must be taken in accordance with environmental factors.

Kraut and Streeter abstracted several characteristics that may affect coordination in software development [2]. *Scale* is a fundamental characteristic of many software systems. If a software system is small, effective coordination can occur because a single individual or small group can direct its work and keep all the implementation details in focus. *Interdependence* is based on the need for integrating thousands of software modules to make them work correctly.

Unlike manufacturing, software development is a nonroutine activity. Zmud noted, "An important insight to understanding the problems associated with managing software development is that most difficulties can be traced to the uncertainty that pervades software development" [9]. *Uncertainty*, the absence of complete information, stems from the complexity of the environment and dynamism, or the frequency of changes to various environmental variables, or state-of-the-art technologies [9][10]. It also increases because specifications of the functionality of the software change over time.

Although many methods have been devised to cope with the combination of large size and interdependence, *informal communication* invariably has a valuable role in consensus formation, information sharing, and other activities for smooth coordination.

2.3. Dimensions for Classifying Projects

Though there may be many aspects for contingency factors, we focused on three dimensions: customer type, technology-orientation, and management style. These dimensions will affect knowledge networks for the following reasons.

1. *Customer Dimension*

Competitive hostility, market turbulence, and the ease of market entry all increase environmental uncertainty. One way to cope with uncertainty is to implement structural (often tight) mechanisms that enhance information flow. Then, if some kind of “match” or “fit” is expected, the more uncertainty, the tighter a knowledge network must be. In contrast, a weak knowledge network will be found when a project faces relatively lower uncertainty.

The customer dimension adds uncertainty to software developments. For example, the size of systems used in government offices is often big, and thus, such projects require many resources, including time, money, many engineers, etc. Since the firms which can offer these resources are limited, market hostility is relatively low. By contrast, private firms require high standards (e.g., low price, high-quality) for developers and often functionality changes for specifications of the system. Also, because the resource constraint is relatively low, many firms can easily enter in this market. Then, uncertainty becomes relatively high, and a tight knowledge network should be expected to confront the uncertainty in the market. This discussion leads to our first hypothesis.

H1: The customer dimension (i.e., government offices or private firms) is associated with the density (tight or loose) of knowledge networks.

2. *Technology-Orientation Dimension*

The technology orientation serves as the foundation for the interest in advanced technology, which refers to the set of beliefs that puts technological interest first, while excluding customer needs. When the target customer is end consumer, because the purposes of using product vary person to person, the requirements for developing systems cannot be easily identified. In this situation, system developers tend to make efforts to equip many functions into the products to meet a variety of customer needs, instead of determining the “true” customer needs.

For this reason, technology-oriented projects do not need to determine customer needs as clearly as demand-pull type projects which offer B-to-B products. This leads to our next hypothesis.

H2: In technology-oriented projects, the density of knowledge networks is relatively loose, and the level of obtaining customer needs is relatively low.

3. *Management Dimension*

It must not be a good assumption that coordination techniques are determined only by external factors. Taking internal factors into consideration, the differences of management style must create an important aspect. The differences may appear in many aspects. For example, when some projects are being done in regional branches at the same firm, whether it is in head office or regional branches must affect the level of customer needs.

H3: The level of obtaining customer needs is affected by whether the project is in head office or regional branches.

3. Survey

3.1. Sample

To identify a knowledge network, we organized a survey at a large Japanese firm which mainly provides system integration services. All employees in this firm were asked to respond to a questionnaire. After excluding data from areas not directly associated with product (system) development such as human resources, we had 1,646 data, corresponding to a response rate of 37.4%.

3.2. Knowledge Network

Each respondent was asked if to obtain information related to customer needs with the following question: Do you have a contact to [the process] to get information related to customer needs? (Yes or No) In this phrase, [the process] means each software development stage: sales, analysis (system analysis), design (system design), code, test, maintenance, and customer as a source of customer needs. Since most of the respondents were in charge of tasks corresponding to two or more processes, the ratio of knowledge flow was calculated by a weighted average by an inverse number of processes overlapping of each respondent.

$$\alpha_{ij} = \sum_{n \in P_j} \frac{t_{n,i}}{h_n} / \sum_n \frac{1}{h_n}$$

α_{ij} : ratio of knowledge flow from process-i to process-j

$t_{n,i}$: contact to process-i by respondent n (1 or 0)

h_n : number of processes overlapping of respondent n

P_j : a set of process-j involved

3.3. Organizational Characteristics

The level of obtaining or reflecting customer needs were measured by asking how well the work groups of respondents actually obtain or reflect customer needs. The correspondence questions are as follows (the response scale is: 1. strongly disagree - 7. strongly agree; on a Likert-Scale).

- (a) obtaining customer needs: “your working group fully obtain customer needs.”
- (b) reflecting customer needs: “your working group fully reflect customer needs in your work.”

From the subtraction between the levels of these variables, we can estimate the level of original effort for embodying customer needs into their work. For example, if the level of reflecting customer needs is higher than that of obtaining customer needs, the respondent is assumed to make his or her own efforts into the work.

3.4. Division Classifications

For our purpose, it is suitable that projects in this firm are classified. Though since so many projects are running in the firm, it is impracticable to identify what projects are under what conditions. Then, we alternatively consider division classifications as shown in figure 1, classifying 16 divisions in this firm into four types to meet our concern.

- (a) Demand-Pull (government offices)
The divisions categorized in this type are offering made-to-order products, and their main customers are government offices. The main concern of customers is not the price or technical advancement but that products work stably. The number of competitors is limited, and thus, market turbulence seems to be relatively low.

- (b) Demand-Pull (private firms)
The concerns of private firms for implementing systems have a wide range of aspects: price, delivery (deadline), quality, etc. Many competitors exist in the market, and thus, hostility between them is fierce. For this reason, it is reasonably assumed environmental uncertainty is higher than that of former type. In reality, the reputation of products this type of division offers is higher than that of other types.

- (c) Technology-Push
Compared to the demand-pull type offering B-to-B products, the divisions in this type mainly offer consumer products (B-to-C products) like packaging softwares. The aim of these technology-oriented divisions for this firm is to pursue brand-new technologies that will be needed or used for future products.

- (d) Regional Branch
The firm we investigated has six branch offices in Japanese regional area (head office is located in Tokyo). The aim of regional branches is to maintain close-ties with customers and deal with their problems or complaints about the systems as soon as possible. They offer a variety of products of demand-pull-type as well as technology-push-type that are also for government offices as well as private firms. All of the branches were initially operated by other firms. Three of them were merged just a few months before our survey was conducted. Other two were merged no more than three years ago. Thus, these branches must have been left old management styles that were originally developed by previous firms.

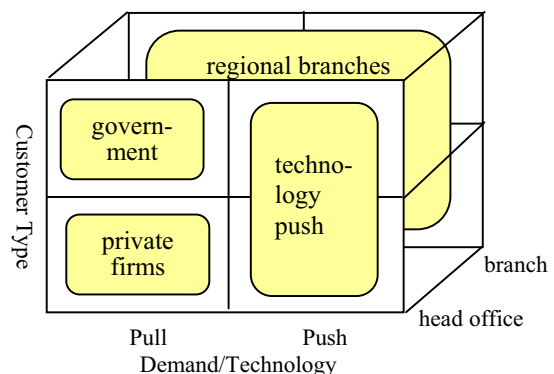


Figure 1. Division Classifications

4. Results

4.1. Case of Demand-Pull (government offices)

Figure 2 shows the extent to which process actually contact to obtain customer needs for each side communication. The arrows indicate the direction of choices to obtain customer needs. Heavy, thin, dashed arrows correspond to the extent of the knowledge flow. The threshold levels that distinguish these arrow types are settled by the average level in top order of 5-6th (heavy-thin), 10-11th (thin-dash), 15-16th (dash-none) knowledge flow using all data. The actual level of each is 0.823, 0.647, 0.540, respectively.

In the figure, each process has direct passes from 'customer', and the structure is distinctly different from the linear-processing model (i.e., water-fall model). It implies that because customer needs is somewhat 'sticky' in itself [11], downward processes directly ask what the real meaning of customer needs is. In addition, some back-flows exist at 'design to analysis' and at 'maintenance to analysis.' Additionally, 'sales' is isolated from other processes, showing that some kind of bottleneck exists between 'sales' and other processes. This is also confirmed in Figure 3, which shows the average level of obtaining and reflecting customer needs in each process. Both levels in 'sales' are very low compared to other processes. Moreover, the level of obtaining customer needs in 'sales' is higher than that of reflecting customer needs while inverse results are confirmed in other processes. This implies that 'sales' does not try to add an original effort into their work as compared with other processes. Maybe it is because that the main concern of sales persons is to maintain close-relationship with customer, rather than to identify the real customer needs or to convey it to other processes.

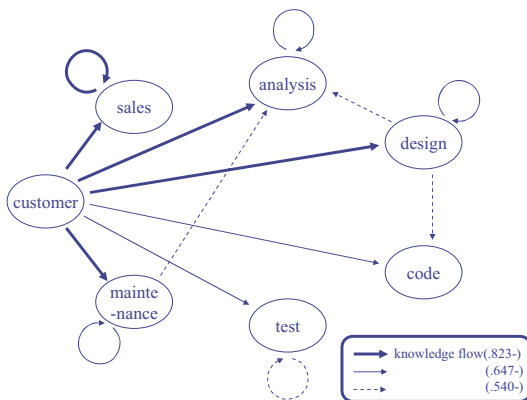


Figure 2. Knowledge Network (government offices)

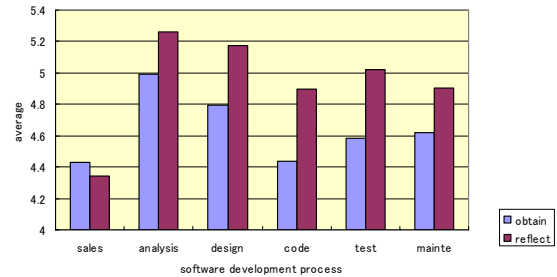


Figure 3. Customer Needs (government offices)

4.2. Case of Demand-Pull (private firms)

Comparing Figure 2 and 4, it is apparent whether customers are government offices or private firms has a great impact on a knowledge network. The knowledge network in this case implies that projects build tighter networks to face much environmental uncertainty. Additionally, both the levels of obtaining and reflecting customer needs in this type are higher than in other types, and the levels at each process are almost the same. Also, the down trend from 'analysis to code' confirmed in Figure 3 does not exist in this case.

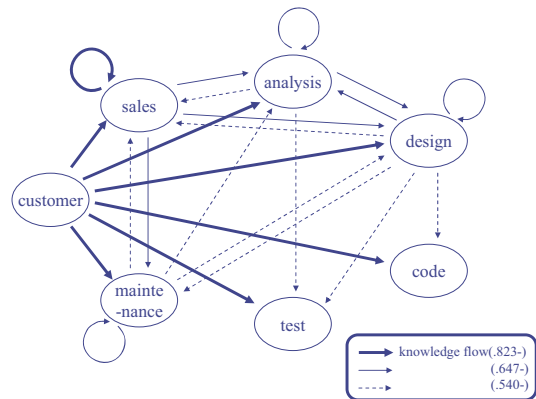


Figure 4. Knowledge Network (private firms)

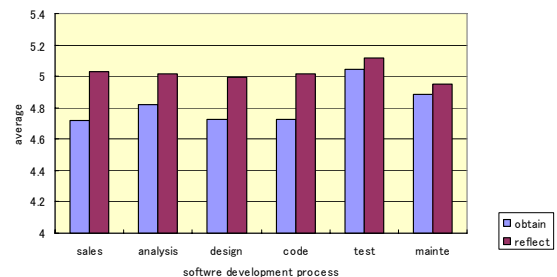


Figure 5. Customer Needs (private firms)

4.3. Case of Technology-Push

The low density in a knowledge network in the case of the divisions of technology-push is caused because they cannot directly ask what products their customers (end consumers) need or want. They tend to rely more on their feelings or experiences rather than meeting with or hearing customers to estimate customer needs. In terms of obtaining customer needs, although the density of a knowledge network is loose, the level of it is estimated relatively higher in this type, which contradicts our hypothesis.

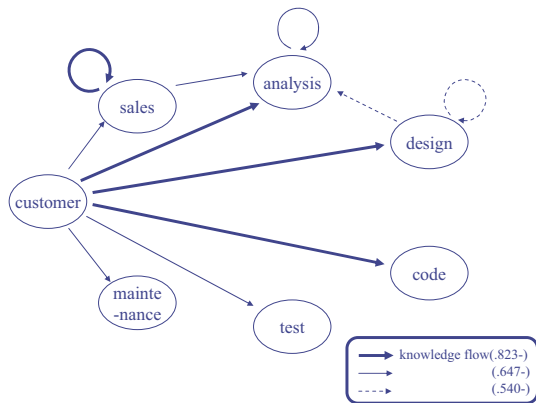


Figure 6. Knowledge Network (technology-push)

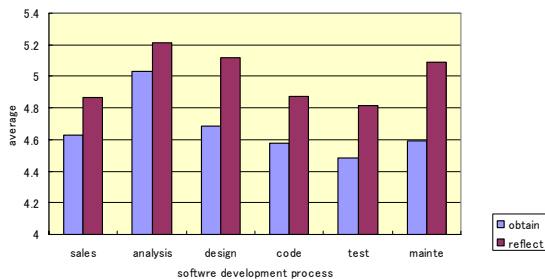


Figure 7. Customer Needs (technology-push)

4.4. Case of Regional Branches

In the case of regional branches, the density of the knowledge network is almost at an average level. However, the level of obtaining and reflecting customer needs in this type is lower compared to that in other cases. This result implies that differences of management style (as we noted, regional branches were initially operated by another firm and must employ differing management styles) impact not on the knowledge network but on the level of obtaining or reflecting customer needs.

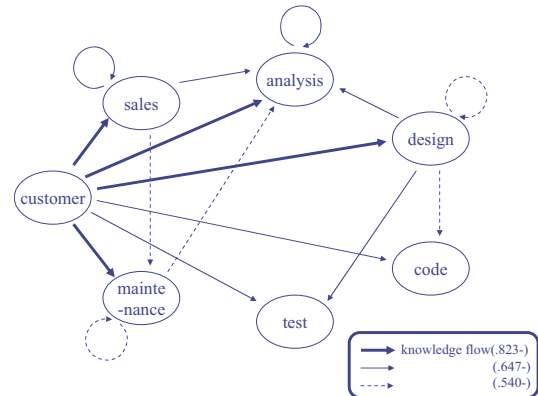


Figure 8. Knowledge Network (regional branches)

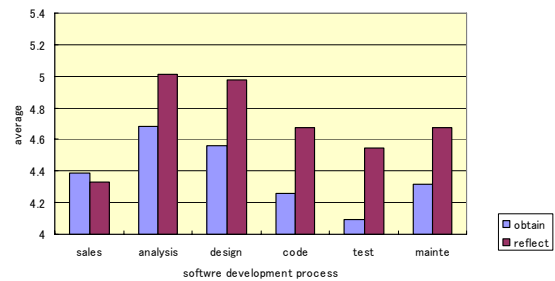


Figure 9. Customer Needs (regional branches)

4.5. Summary

The results are summarized in Table 2. We knew the customer type affects the density or structure of the knowledge network, supporting H1. Additionally, the knowledge network is affected by the technology-orientation, though the level of obtaining or reflecting customer needs is not affected so strongly (H2 is partly supported). It is assumed that employees in this type must strongly rely on their own ideas for determining customer needs. In addition, the supposable management differences (i.e., head office or regional branches) relate to the level of obtaining or reflecting customer needs (H3 is supported). It implies that there may be other organizational factors that determine the level of customer needs than the knowledge network.

Table 2. Summary of Results

	demand-pull (government)	demand-pull (private firms)	technology push	regional branches
knowledge network	average	tight	loose	average
level of customer needs	average	high and even	average	low

5. Discussion

Taking a contingency view into account, the research problem becomes to be identifying the structure that maximizes performances for a given environment. Our first assumption is that the knowledge network must be the most determinative factor for the level of customer needs. Viewing knowledge networks under some conditions, our results show that it is partly true but there may be other factors that impede or foster obtaining or reflecting customer needs. Taking our results into consideration, focusing on organizational ability or culture is thinkable factor to interpret the background of this problem.

Some scholars have pointed out that organizational ability creates the basis for obtaining customer needs. Cohen and Levinthal argued that firms need absorptive capacity: the ability to recognize the value of new, external information, assimilate it, and apply it to commercial ends [12]. Kogut and Zander proposed a concept of combinative capability, which refers to the capacity of a firm to combine and recombine existing knowledge [13]. Related arguments have been discussed by many scholars [11][14][15]. Taking the tacit dimension in customer needs into consideration, the ability of abstracting meanings of customer needs through personal interactions may be a strong factor for determining the level of obtaining customer needs.

On the other hand, there is a standpoint that focuses on organizational culture or climate, that puts customer's interest first, while excluding those of other stakeholders such as owners, managers, and employees, in order to develop a long-term profitable enterprise [16]. A simple focus on information about the needs of actual and potential customers is inadequate without consideration of the more deeply rooted set of values and beliefs that are likely to consistently reinforce such a customer focus and pervade the organization. For example, Deshpande, Farley, and Webster noted that such a belief can be achieved only if it is complemented by a spirit of entrepreneurship and an appropriate organizational climate [17]. It is also considered as manifest in many aspects of organizational performance, and then, constructing such an organizational customer or culture must be the key antecedents of obtaining customer needs.

Whatever the standpoints are, organizational abilities or cultures cannot be established in just a few years. In our analysis, the result in the case of regional branches implies that. Although a merger activity was done, the level of obtaining customer needs cannot be enhanced so rapidly. It is also

expected that changing a knowledge network in a software project also takes considerable time. We therefore, had better to think that a long-time view is needed to take an action in this problem.

6. Conclusion

The primary concern in this paper was to investigate the knowledge network, and to determine how and to what extent it relates to the level of obtaining or reflecting customer needs. Although we do not prove into the mechanisms between them in detail, several results are worth highlighting.

The first point is that structures of the knowledge networks are complex and not like a linear-processing model (i.e., water-fall model). It suggests that customers have to show (or to be asked) their needs or wants to many processes. This implies that customer needs are sticky and cannot easily to be absorbed into a firm [11]. Taking the tacit dimension in customer needs into consideration, the ability to convert customer needs (often in tacit dimension) into software requirements (often in explicit dimension) is a central concern to attain effective network within a project.

Second, our analysis showed that the structure or density of a knowledge network is strongly affected by the environmental factors that each project faces. This could be caused by environmental uncertainty or technical orientation, and other factors. In addition, the knowledge network is a strong antecedent of obtaining customer needs, however, not a determining factor. Our analysis showed the level of obtaining customer needs is affected not only by knowledge networks but also by some other organizational abilities, such as absorptive capacity, organizational culture or climate, etc. It implies that when we want to build an effective knowledge network in a project, many factors must to be taken into consideration.

In this paper, we have focused only on the problem of knowledge networks or obtaining customer needs. Naturally, there must be other concerns to build an effective coordination or collaboration in software development. Nonetheless, the importance of personal interactions will have been a central issue. Until now, many researchers have pointed out the importance of this issue, however, in our view, it is not just a matter of the frequency of interaction but of careful coordination with environmental factors and organizational abilities. When we want to attain an effective collaborative works in software development, it is recommendable not to underestimate many aspects which we have taken up in this paper.

References

- [1] Van de Ven, A. H., Delbecq, A. L., and Koenig, R. Jr(1976), "Determinants of Coordination Modes within Organizations," *American Sociological Review*, Vol.41, No.2, pp.322-338.
- [2] Kraut, R. E. and Streeter, L. A.(1995), "Coordination in Software Development," *Communications of the ACM*, Vol.38, No.3, pp.69-83.
- [3] Kim, K. K. and Umanath, N.(1992), "Structure and Perceived Effectiveness of Software Development Subunits: A Task Contingency Analysis," *Journal of Management Information Systems*, Vol.9, No.3, pp.157-181.
- [4] Premkumar, G. Ramamurthy, K., and Saunders, C. S.(2005), "Information Processing View of Organizations: An exploratory Examination of Fit in the Context of Interorganizational Relationships," *Journal of Management Information Systems*, Vol.22, No.1, pp.257-294.
- [5] Sabherwal, R. (2003), "The Evolution of Coordination in Outsourced Software Development Projects: A Comparison of Client and Vendor Perspectives," *Information and Organization*, Vol.13, No.3, pp.153-202.
- [6] Andres, H. P. and Zmud, R. W. (2001), "A Contingency Approach to Software Project Coordination," *Journal of Management Information Systems*, Vol.18, No.3, pp.41-70.
- [7] Lawrence, P. R. and Lorsch, J. W.(1967), *Organization and Environment: Managing Differentiation and Integration*, Harvard Business School, Division of Research.
- [8] Scott, W. R.(2003), *Organizations: Rational, Natural, and Open Systems(5th eds)*, Prentice Hall.
- [9] Zmud, R.W. (1980), "Management of Large Software Development Efforts," *MIS Quarterly*, Vol.4 (June), pp.45-55.
- [10] Duncan, R.E.(1972), "Characteristics of Organizational Environments and Perceived Environmental Uncertainty," *Administrative Science Quarterly*, Vol.17, No.3, pp.313-327.
- [11] von Hippel, E.(1994), "'Sticky Information" and the Locus of Problem Solving: Implications for Innovation," *Management Science*, Vol.40, No.4 (April), pp.429-439.
- [12] Cohen, W. M. and Levinthal, D. A. (1990), "Absorptive Capacity: A New Perspective on Learning and Innovation," *Administrative Science Quarterly*, Vol.35, pp.128-152.
- [13] Kogut, B. and U. Zander(1992). "Knowledge of the Firm: Combinative Capabilities, and the Replication of the Firm," *Organization Science*, Vol.3, No.3, pp.383-397.
- [14] Teece, D.J.(1977), "Technology Transfer by Multinational Firms: the Resource Cost of Transferring Technological Know-How," *Economic Journal*, Vol.8 (June), pp.242-261.
- [15] Szulanski, G.(1996). "Exploring Internal Stickiness: Implements to the Transfer of Best Practice Within the Firm," *Strategic Management Journal*, Vol.17 (Winter Special Issue), pp.27-43.
- [16] Hurlley, R.F. and Hult, G.T.M.(1999), "Innovation, Market Orientation, and Organizational Learning: An Integration and Empirical Examination," *Journal of Marketing*, Vol.62 (July), pp.42-54.
- [17] Deshpande, R., Farley, J. U. and Webster, Jr, F. E.(1993), "Corporate Culture, Customer Orientation, and Innovativeness in Japanese Firms: A Quadrad Analysis," *Journal of Marketing*, Vol.57 (January), pp.23-37.

Coordinating Multi-Team Variability Modeling in Product Line Engineering

Deepak Dhungana Rick Rabiser Paul Grünbacher
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University, 4040 Linz, Austria
{dhungana, rabiser}@ase.jku.at paul.gruenbacher@jku.at

Abstract

In product line engineering variability models capture the commonalities and variability of core assets and guide product derivation. In large-scale systems the knowledge that is required for creating and evolving variability models is typically distributed among different heterogeneous stakeholders. For example, sales people usually think in terms of features and monetary resources while developers emphasize architectural elements, software resources, and configuration parameters. This paper is based on experiences from an ongoing industrial research project and proposes an approach for sharing variability knowledge in a multi-team development organization. Our approach allows different teams to create a variability model from their point of view (e.g., for a subsystem they are responsible for). Subsequently all created models are combined to one integrated variability model.

1. Introduction

It has been demonstrated that product line engineering (PLE) can reduce cost and increase productivity and quality through consequent reuse of core assets [8]. Variability management is a key concept in PLE to express the commonalities and variability of core assets and to understand dependencies among them [4]. Variability models can also be seen as building plans for product instantiation and configuration.

Creating a variability model is not trivial as it relies on information spread across the minds of numerous heterogeneous stakeholders. Today's software systems are large and often different development teams are in charge for different parts of the system. The dependencies between the communication structure of a development team and the technical structure of a system have been addressed by Conway's law [1, 5]. Working with large-scale variability models requires mechanisms that support the cooperation of different teams

building and evolving such models for the parts of the system they are dealing with. We are facing these challenges in an ongoing research project with Siemens VAI, the world leader in engineering and building plants for the iron, steel, and aluminum industries. In this paper we propose an approach for distributed editing and integration of variability knowledge.

2. Capturing Variability Knowledge

As part of our research in bridging the gap between stakeholders in PLE, we have been developing an integrated variability model [3]. Our model covers product line assets (e.g., components, resources, features) and decisions as shown in Figure 1.

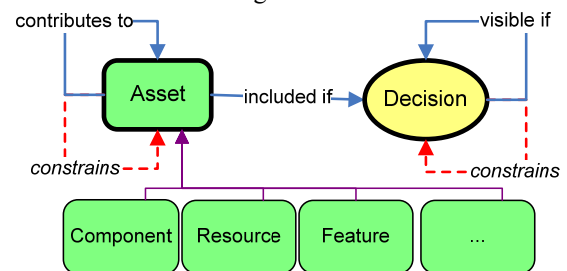


Figure 1: Integrated variability model

Assets address different levels of variability including customer visible properties expressed as features, architectural elements such as components, and implementation level details such as properties. Assets can have structural (*contributes to*) and logical (*constrains*) dependencies. Decisions are used to link the various model elements. A decision can be seen as a variation point, where the user is given an option. The decision taken by a user influences the selection of assets. Decisions can also have dependencies. *Visible if* models the situation that the selection of a certain value makes another decision relevant. Decision can also constrain other decisions.

While this model works well for capturing different PLE aspects such as customer-oriented perspectives

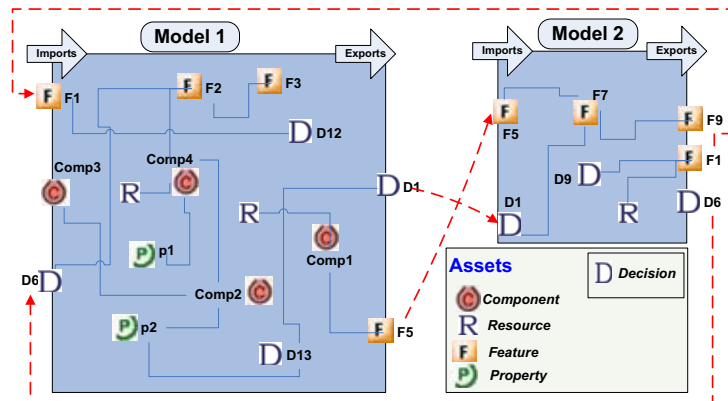


Figure 2: Sharing and merging of variability models

and technical perspectives, it is unrealistic to assume that such a model can be created and evolved by an individual or by a small team for a real-world system. Interaction with our industry partner confirmed that teams typically have detailed knowledge of a small set of subsystems and only rough knowledge about other subsystems [2]. It is therefore important to support the interaction of different teams via distributed variability models linked through model connectors [7] to allow teams to separately create and evolve models. Consistency is ensured through the defined model connectors which allow combining all individual variability models into one integrated variability model.

3. Sharing and Merging Variability Models

Our approach is based on concepts in architecture description languages (ADLs) [6]. Such languages support modeling of large-scale systems by defining interfaces of subsystems and the interaction of subsystems. While ADLs focus on architectural elements such as components or connectors we extend this idea to all elements of our integrated variability model. A team working on a variability model can specify elements of the model (see Figure 1) as public to “export” them. Variability models of other subsystems can then “import” the public elements as part of their own variability model (e.g., when specifying constraints between subsystems). Private elements are internal to a subsystem with no relationships to elements in other models. Distributed teams building individual models only have to know what other elements have an effect on the elements they create. It is not necessary (and often not possible) to know what effect a newly added element has on elements in other models. When introducing a new component to a model, it is easier to describe which other components and resources are

needed by this component, rather than modeling where this new component can be used.

The mapping between exported elements of one model and imported elements of another model can be automated. A match can be found for elements of the same type and with the same name. As illustrated in Figure 2, *model 1* exports a decision *d1* and a feature *f5*, which are imported by *model 2*. By merging the individual models a complete variability model is created that guides the instantiation of products in PLE. We have been developing an initial prototype of this capability as part of our ongoing tool development.

References

- [1] M.E. Conway, “How Do Committees invent?”, *Data-mation*, 14 (4), 1968, pp. 28–31.
- [2] D. Dhungana, R. Rabiser, P. Grünbacher, H. Prähofer, C. Federspiel, and K. Lehner, „Architectural Knowledge in Product Line Engineering: An Industrial Case Study“, *Proc. 32nd Euromicro Conf. on Software Engineering and Advanced Applications*, IEEE CS, 2006.
- [3] D. Dhungana, “Integrated Variability Modeling of Features and Architecture in Software Product Line Engineering”, *Doctoral Symposium, 21st IEEE/ACM Int. Conf. on Automated Software Engineering*, Tokyo, Japan, 2006.
- [4] J. Estublier and G. Vega, “Reuse and Variability in Large Software Applications”, *Proc. 10th European Software Engineering Conference*, Portugal, 2005, pp. 316–325.
- [5] J.D. Herbsleb and R.E. Grinter, “Architectures, Coordination, and Distance: Conway’s Law and Beyond”. *IEEE Software*, 16 (5), 1999, pp. 63–70.
- [6] N. Medvidovic and R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, *IEEE TSE*, 26 (1), 2000, pp. 70–93.
- [7] N. Medvidovic, P. Grünbacher, A.F. Egyed, and B.W. Boehm, “Bridging Models across the Software Lifecycle”, *Journal of Systems and Software*, 68 (3), 2003, pp. 199–215.
- [8] K. Pohl, G. Böckle, and F.J. van der Linden, “*Software Product Line Engineering: Foundations, Principles and Techniques*”, Springer-Verlag, 2005.

Learning Support by Reflection and Knowledge Collaboration in a Team-based Software Engineering Project Course - Position paper -

Atsuo HAZEYAMA

Department of Information Science, Tokyo Gakugei University

E-mail: hazeyama@u-gakugei.ac.jp

Abstract

Software development is a highly knowledge-intensive and collaborative activity. Problem resolution processes are performed iteratively during software development. We propose a learning model that is based on reflection and knowledge collaboration for problem resolution in a software engineering project course. We also describe an overview of a support system based on this model.

1. Introduction

Software development is a highly knowledge-intensive [8] and collaborative activity [1]. Problem resolution processes are performed iteratively during software development. Individual developers possess only a specific part of the knowledge and expertise required for software development. Therefore, they develop software and simultaneously collect various types of information. Some developers even consult experts [9]. Ye emphasizes the importance of knowledge collaboration in software engineering [9]. For this purpose, he provides the following three types of support facilities for a social platform: finding sample programs, browsing the archives of previous discussions, and posing questions to selected experts.

We tackle studies on a team-based software engineering project course [3]. There it is important for learners to acquire knowledge and skills on software development. I propose to introduce reflection support as well as knowledge collaboration for the domain. Reid defined reflection as being a process of reviewing an experience of practice in order to describe, analyze, evaluate and so inform learning about practice [6]. Sample programs, discussions with respect to past problem solving, and/or answers to questions will be important information for a learner's problem solving. In addition, I believe reflection of his/her problem solving and recording of the process enhance his/her understanding. These processes correspond to internalization and externalization of the SECI model [5]. Furthermore by sharing the information that was described as the result of reflection, it may be able to contribute to problem solving of other learners. We regard problem solving that occurs during software

development as a kind of learning. Therefore I think it is also useful for software engineers to describe their reflection process in professional software development.

Hazzan described significance of introducing reflective perspective [7] and studio concept into software engineering education [4]. That paper described the process, which facilitates reflection. However, it did not clarify how results of reflection are externalized and the supporting environment.

2. Proposal

This section proposes a conceptual framework for learning support by reflection and knowledge collaboration in a team-based software engineering project course.

2.1 Conceptual model

When a person encounters a problem, the following three problem solving patterns are considered:

- (1) Solving the problem by himself/herself
- (2) Solving the problem over searching for related information and referring to it
- (3) Solving the problem by posing questions to others

“To present sample programs” and “to browse past archived discussions” out of the three facilities Ye provided correspond to (2). “To pose questions to selected experts” correspond to (3). This study supports the abovementioned three patterns. Even if which pattern is adopted, I ask the learner for reflection and describing the result after problem resolution. Through the process, I aim at enhancing his/her knowledge. Especially when a learner solved his/her problem by referring to past archived discussions and/or sample programs other created, or by advices from other learners, reflection improves his/her understanding and it enhances the information the learner referred. This process corresponds to combination of the SECI model. This information is valuable when another learner reuses it.

Figure 1 shows information structure for learning support by reflection and knowledge collaboration. It is consisted of five major objects, the problem

concerned and description by reflection for it, target artifacts that included the problem and that the problem has been fixed, external resources that were referred for problem solving, discussion archives with others, and other problems that were related to the concerned problem. We adopted Shippaigaku's attributes for reflection [2]. Shippaigaku is a theory whose goal is to learn from failures. It aims at avoiding similar mistakes and/or accidents to past ones by learning. Shippaigaku defines six attributes; a problem description (event) and its accompanied descriptions (background and progress), the information toward problem resolution (cause and disposition) and lessons learned from the problem solving. In addition to the six attributes of Shippaigaku, I manage the following information and associate them with the problem-solving information: the result of disposition as the target artifact object, external resources, discussion archives, and other related problems.

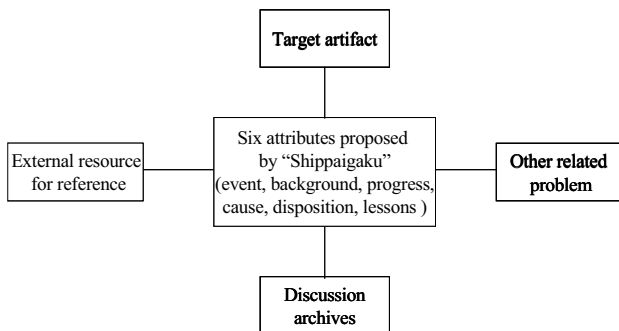


Figure 1. Information structure for our learning support environment

2.2 Knowledge sharing support

As the volume of knowledge accumulated in the environment increases, it is difficult to retrieve appropriate information in an efficient manner. Most knowledge management systems utilize voting information and/or access log information for this purpose [10]. In addition, I associate a problem with usecase, which is unit of a function a system has because we suppose similar tasks may have similar problems. We use a usecase name in order to specify functionality.

3. Summary

We have proposed a learning model for the problem resolution process in a team-based software engineering project course; this model integrates reflection with knowledge collaboration. We will implement this model and apply it to an actual software engineering project course to validate the proposed framework. We will also evaluate a

trade-off problem between the cost to describe the reflection process and learning effectiveness.

Acknowledgments

The author would like to thank anonymous reviewers for their comments to improve this paper. This study is supported by the Grant-in Aid for No. (C) 18500701 from The Ministry of Education, Science, Sports and Culture of Japan.

References

- [1] D. M. German, D. Cubranic, and M-A. D. Storey, "A Framework for Describing and Understanding Mining Tools in Software Development", *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR 2005)*, pp. 95-99, ACM Press, 2005.
- [2] Y. Hatamura, *Shippaigaku no susume*, Kodansha, 2000 (in Japanese).
- [3] A. Hazeyama, "An Education Class on Design and Implementation of an Information System in a University and Its Evaluation", *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, IEEE CS Press, pp. 21 - 27, October 2000.
- [4] O. Hazzan, "The reflective practitioner perspective in software engineering education", *The Journal of Systems and Software*, Vol. 63, pp. 161-171, 2002.
- [5] I. Nonaka, and H. Takeuchi, *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, 1995.
- [6] B. Reid, "But we're doing it already" Exploring a response to the concept of reflective practice in order to improve its facilitation. *Nurse Ed Today*, Vol. 13, pp. 305-309, 1993.
- [7] D. A. Schon, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, 1983.
- [8] Y. Ye, and K. Kishida, 1st International Workshop on Supporting Knowledge Collaboration in Software Development, in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC2005)*, 2005, <http://13d.cs.colorado.edu/%7Eyunwen/KCSD2005/>.
- [9] Y. Ye, "Socio-Technical Support for Knowledge Collaboration in Software Development Tools", *Proceedings of the Workshop on Integrating Software Engineering and Usability*, pp. 39-51, 2005.
- [10] M. Zacklad, "Communities of action: a cognitive and social approach to the design of CSCW systems", *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, pp. 190 - 197, ACM Press, 2003.

Knowledge Collaboration by Mining Software Repositories

Thomas Zimmermann
Saarland University, Saarbrücken, Germany
tz@acm.org

Abstract

We will give a short overview on recent approaches to support developers by mining software repositories and outline current and future challenges from which knowledge collaboration can benefit.

1. Introduction

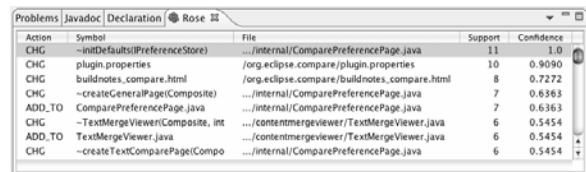
When people collaborate, they communicate and create documents that are shared among each other. In most projects these artifacts are collected and archived in software repositories: For open source projects, communications between developers are stored in *mailing lists*, *newsgroups*, and *personal archives*. Changes to the source code of software are recorded in *version archives* such as CVS. Failures and feature requests are submitted to and discussed in *issue tracking systems* such as Bugzilla. Explicit knowledge such as documentation and design documents is published on websites or wikis.

Recently a new research area evolved that mines software repositories. Although most approaches have focused on understanding software and its evolution so far, software repositories can be leveraged to support developers and their collaboration.

In this paper, we will give a short overview on the state-of-art of mining software repositories with respect to collaboration (Section 2), before we outline ongoing and future challenges from which knowledge collaboration can benefit (Section 3).

2. Supporting Developers

In this section we present several examples how historic data was used to support collaboration among developers. Our overview is not complete since we favored research that actually resulted in tools. For a broader view on mining software repositories we refer to the MSR workshop series [6].



Action	Symbol	File	Support	Confidence
CHG	--initDefaults(IPreferenceStore)	.../internal/ComparePreferencePage.java	11	1.0
CHG	plugin.properties	/org.eclipse.compare/plugin.properties	10	0.9090
CHG	buildnotes_compare.html	/org.eclipse.compare/buildnotes_compare.html	8	0.7272
CHG	--createGeneralPage(Composite)	.../internal/ComparePreferencePage.java	7	0.8363
ADD_TO	ComparePreferencePage.java	.../internal/ComparePreferencePage.java	7	0.6363
CHG	--TextMergeViewer(Composite, int)	.../contentmergeviewer/TextMergeViewer.java	6	0.5454
ADD_TO	TextMergeViewer.java	.../contentmergeviewer/TextMergeViewer.java	6	0.5454
CHG	--createTextComparePage(Compo	.../internal/ComparePreferencePage.java	6	0.5454

Figure 1. After an initial change to a method, eROSE recommends related code locations.

Project memory. The Hipikat tool by Cubranic et al. [2] was the first one to combine artifacts from different software repositories such as version archives, bug databases, documentation, and mailing lists. Developers can explicitly query this project *memory* for related artifacts after selecting an initial artifact. Hipikat’s recommendations are especially useful for newcomers to a software project.

Guiding developers. The eROSE tool by Zimmermann et al. [10] guides programmers along related changes by mining version archives. When a developer changes $f()$ and other people have changed $f()$ together with $g()$ in the past, eROSE will detect this and suggest “*Programmers who changed function $f()$ also changed function $g()$* ” (see Figure 1). In contrast to Hipikat, eROSE makes recommendations automatically and suggests specific actions (change, add, or delete something).

Software navigation. The NavTracks tool by Singer et al. [7] monitors the *navigation history* of a single developer and use this data to support her future navigation. DeLine et al. [3] extended this work in their Team Tracks tool to multiple developers that share navigation history.

All these tools leverage one or more software repositories to support developers by providing knowledge that is obtained from the past. In the next section, we will outline ongoing research challenges that will further improve knowledge collaboration.

3. Challenges

The research on mining software repositories is currently in an early stage. There are several ongoing challenges that are relevant for knowledge collaboration.

Multiple data sources. Most research focuses only on one data source such as version archives or bug databases. In recent research several software repositories have been combined (starting with Hipikat [2]). This gives additional context to mining. For instance, one can assess changes using bug databases, thus getting a notion of good vs. bad knowledge.

Fine-grained changes. All tools discussed in Section 2 focused only on artifact level such as files, methods, or bug reports. Recently, more fine-grained changes were analyzed [4] and used to identify usage patterns [5] or cross-cutting concerns [1]. Combined with context information this will lead to tools that can assess new changes based on knowledge that is mined from software repositories (think of a self-learning bad smell check across developers).

Collecting new data. Most research analyzed existing software repositories. However, at some point the information available will be exhausted. The NavTracks [7] and Team Tracks [3] tools pioneered a new direction. Instead of taking existing repositories they build their own repositories which are then analyzed. This way, one gets more and better data to turn into knowledge. Related research in this area includes waypointing and social tagging of software as proposed by Storey et al. [8].

Mining across projects. Typically multiple projects are mined at the same time for understanding software evolution. However, when it comes to supporting developer, only single projects were investigated so far. Xie and Pei were the first ones to mine knowledge (usage patterns) across multiple projects [9]. By considering a large amount of projects, one can build a huge knowledge base. The goal will be to improve search engines for source code such as Kodiers¹ and smoothly integrate them into IDEs.

Although mining software repositories does not explicitly support collaboration, it creates knowledge that helps developers. Since this knowledge is mined from data that comes from different developers, one can think of *implicit* knowledge collaboration: the knowledge is collected in the background and shared among developers.

¹ <http://www.koders.com/>

4. References

- [1] Silvia Breu and Thomas Zimmermann. "Mining Aspects from History." In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), September 2006.
- [2] Davor Cubranic, Gail C. Murphy, Janice Singer, Kellogg S. Booth. "Hipikat: A Project Memory for Software Development." In *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446-465, June 2005.
- [3] Robert DeLine, Mary Czerwinski, George G. Robertson. "Easing Program Comprehension by Sharing Navigation Data." In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*, September 2005, Dallas, USA. IEEE Computer Society, pp. 241-248
- [4] Beat Fluri and Harald C. Gall. "Classifying Change Types for Qualifying Change Couplings." In *Proceedings of the International Conference on Program Comprehension (ICPC)*, Athens, Greece, June 2006, pp. 35-45.
- [5] V. Benjamin Livshits and Thomas Zimmermann. "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories." In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2005)*, Lisbon, Portugal, September 2005, pp. 296-305.
- [6] International Workshop on Mining Software Repositories 2004-2006, <http://msr.uwaterloo.ca/>
- [7] Janice Singer, Robert Elves, Margaret-Anne Storey. "NavTracks: Supporting Navigation in Software Maintenance." In *Proceedings 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 325-334, September 2005.
- [8] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. "Waypointing and social tagging to support program navigation." In *CHI '06: Extended Abstracts on Human Factors in Computing Systems*. Montréal, Québec, Canada, April 2006. ACM Press, New York, NY, pp. 1367-1372.
- [9] Tao Xie and Jian Pei. "MAPO: mining API usages from open source repositories." In *Proceedings of the International Workshop on Mining Software Repositories (MSR '06)*, Shanghai, China, May 2006. ACM Press, New York, NY, pp. 54-57.
- [10] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, Andreas Zeller. "Mining Version Histories to Guide Software Changes." In *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, June 2005.