# Proceedings of The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009)

*hosted by the first JSAI International Symposia on AI*

Workshop Co-Chair

Masao Ohira & Yunwen Ye

Campus Innovation Center, Tokyo, Japan,
November 19-20, 2009

ISBN 4-915905-39-X C3004(JSAI)

Proceedings of the

**3rd International Workshop on**

# Supporting Knowledge Collaboration in Software Development

**KCSD2009**

November 19-20, 2009
Tokyo, Japan

In association with
the First JSAI International Symposia on
Artificial Intelligence (JSAI-isAI2009)

Editors:
Masao Ohira & Yunwen Ye

## Table of Contents

Workshop program…....................................................................................................................v

Workshop organization...............................................................................................................vi

**Keynote address:**

**Knowledge Collaboration in Distributed Software Development** .................................1

***André van der Hoek (University of California, USA)***

**Session 1: Practices**

Barriers to Sharing Domain Knowledge in Software Development Practice in SMEs.........................2
*Jim Buchan, Christian Harsana Ekadharmawan and Stephen G. MacDonell (Auckland University of Technology, New Zealand)*

(Short Paper)
Differences of Time between Modification and Re-modification: ....................................................17
An Analysis of a Bug Tracking System
*Akinori Ihara, Masao Ohira, Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan)*

Usage Result of Problem Resolution Information Sharing System for .............................................23
a Software Engineering Course
*Atsuo Hazeyama, Kazuyuki Shimada and Yusuke Kobayashi (Tokyo Gakugei University, Japan)*

**Session 2: Communication Media**

On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study.....................34
*Emad Shihab, Nicolas Bettenburg, Bram Adams and Ahmed E. Hassan (Queen's University, Canada)*

A Time-Lag Analysis toward Improving the Efficiency of Communications ...................................49
among OSS Developers
*Masao Ohira, Kiwako Koyama, Akinori Ihara, Shisuke Matsumoto, Yasutaka Kamei, Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan)*

A Case Study on the Impact of Global Participation on ....................................................................63
Mailing Lists Communications of Open Source Projects
*Ran Tang, Ahmed E. Hassan and Ying Zou (Queen's University, Canada)*

**Keynote address:**
**Understanding Networked Collaboration**..........................................................................77
***Shuichiro Yamamoto (NTT Data Corporation, Japan)***

**Session 3: Support Tools**

(Short Paper)
Identifying the concepts that are searchable with keywords in code search engines..........................78
*Toshihiro Kamiya (National Institute of Advanced Industrial Science and Technology, Japan)*

(Short Paper)
DesignMinders: A Design Knowledge Collaboration Approach........................................................84
*Gerald Bortis and André van der Hoek (University of California, Irvine, USA)*

On the Use of Emerging Design as a Basis for Knowledge Collaboration........................................91
*Tiago Proenca, Nilmax Moura and André van der Hoek (University of California, Irvine, USA)*

**Session 4: Practices**

A Proposal of TIE Model for Communication in Software Development process...........................100
*Masakazu Kanbe, Shuichiro Yamamoto (NTT Data Corporation, Japan) and Toshizumi Ohta (University of Electro-Communications, Japan)*

Comparison of Coordination Communication and Expertise Communication in............................112
Software Development: Their Motives, Characteristics and Needs
*Kumiyo Nakakoji (University of Tokyo & SRA-KTL, Japan), Yunwen Ye (SRA, Inc., Japan) and Yasuhiro Yamamoto (University of Tokyo, Japan)*

# The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009)

| | |
|---|---|
| Date | November 19-20, 2009 |
| Place | Campus Innovation Center, 3-3-6 Shibaura, Minato-ku, Tokyo, Japan 108-0023 |
| Long paper | 30 minites presentation and discussion |
| Short paper | 20 minites presentation and discussion |
| **Nov. 19 (Thu)** | **10:00-10:30 Registration** |
| | **10:30-11:00 Opening: Workshop goal, agenda and self introduction** |
| | **11:00-12:00 Keynote address** |
| | Knowledge Collaboration in Distributed Software Development |
| | André van der Hoek (University of California, USA) |
| | **12:00-14:00 Lunch** |
| | **14:00-15:20 Session 1: Practices** |
| | Barriers to Sharing Domain Knowledge in Software Development Practice in SMEs |
| | Jim Buchan, Christian Harsana Ekadharmawan and Stephen G. MacDonell (Auckland University of Technology) |
| | (Short Paper) |
| | Differences of Time between Modification and Re-modification: An Analysis of a Bug Tracking System |
| | Akinori Ihara, Masao Ohira, Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan) |
| | Usage Result of Problem Resolution Information Sharing System for a Software Engineering Course |
| | Atsuo Hazeyama, Kazuyuki Shimada and Yusuke Kobayashi (Tokyo Gakugei University) |
| | **15:20-15:50 Coffee Break** |
| | **15:50-17:20 Session 2: Communication Media** |
| | On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study |
| | Emad Shihab, Nicolas Bettenburg, Bram Adams and Ahmed E. Hassan (Queen's University, Canada) |
| | A Time-Lag Analysis toward Improving the Efficiency of Communications among OSS Developers |
| | Masao Ohira, Kiwako Koyama, Akinori Ihara, Shisuke Matsumoto, Yasutaka Kamei, Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan) |
| | A Case Study on the Impact of Global Participation on Mailing Lists Communications of Open Source Projects |
| | Ran Tang, Ahmed E. Hassan and Ying Zou (Queen's University, Canada) |
| | |
| **Nov. 20 (Fri)** | **10:00-11:00 Keynote address** |
| | Understanding Networked Collaboration |
| | Shuichiro Yamamoto (NTT Data Corporation, Japan) |
| | **11:00-11:20 Coffee Break** |
| | **11:20-12:30 Session 4: Support Tools** |
| | (Short Paper) |
| | Identifying the concepts that are searchable with keywords in code search engines |
| | Toshihiro Kamiya (National Institute of Advanced Industrial Science and Technology, Japan) |
| | (Short Paper) |
| | DesignMinders: A Design Knowledge Collaboration Approach |
| | Gerald Bortis and André van der Hoek (University of California, Irvine, USA) |
| | On the Use of Emerging Design as a Basis for Knowledge Collaboration |
| | Tiago Proenca, Nilmax Moura and André van der Hoek (University of California, Irvine, USA) |
| | **12:30-14:30 Lunch** |
| | **14:30-15:30 Session 3: Model and Framework** |
| | A Proposal of TIE Model for Communication in Software Development process |
| | Masakazu Kanbe, Shuichiro Yamamoto (NTT Data Corporation, Japan) and Toshizumi Ohta (University of Electro-Communications, Japan) |
| | Comparison of Coordination Communication and Expertise Communication in Software Development: Their Motives, Characteristics and Needs |
| | Kumiyo Nakakoji (University of Tokyo & SRA-KTL, Japan), Yunwen Ye (SRA, Inc., Japan ) and Yasuhiro Yamamoto (University of Tokyo, Japan) |
| | **15:30-15:45 Closing** |

## Workshop Organization

**Workshop Co-Chairs**

Masao Ohira (Nara Institute of Science and Technology, Japan)

Yunwen Ye (Software Research Associates, Inc., Japan)

**Program Committee**

Daniela Fogli (University of Brescia, Italy)

Mark Grechanik, (Accenture Technology Labs / University of Illinois, USA)

André van der Hoek (University of California, USA)

Reid Holmes (University of Washington, USA)

Katsuro Inoue (Osaka University, Japan)

Yasutaka Kamei (Nara Institute of Science and Technology, Japan)

Ken-ichi Matsumoto (Nara Institute of Science and Technology, Japan)

Kumiyo Nakakoji (SRA-KTL Inc., / University of Tokyo, Japan)

Cleidson de Souza (Federal University of Para, Brazil)

Thomas Zimmermann (Microsoft Research, USA)

## Keynote address:

# Knowledge Collaboration in Distributed Software Development

*André van der Hoek (University of California, USA)*

**ABSTRACT**

Knowledge collaboration takes many forms in software development, with numerous tools that have been developed to support the activity. In a distributed setting, knowledge sharing becomes particularly challenging: affordances that exist in a central setting are no longer available in the distributed environment. In this talk, we introduce a theoretical framework that helps us categorize and understand distributed software collaboration tools, use exemplary systems to illustrate what kinds of knowledge they offload from the developers and what advanced forms of knowledge sharing they thereby enable, and provide a roadmap towards future knowledge collaboration tools.

# Barriers to Sharing Domain Knowledge in Software Development Practice in SMEs

Jim Buchan[1], Christian Harsana Ekadharmawan[1]
[1] School of Computing and Mathematical Sciences,
Auckland University of Technology
Auckland, New Zealand
jim.buchan@aut.ac.nz, christian.harsana@gmail.com

**Abstract.** The collaborative development of a shared domain understanding between the client stakeholders and the software production team is crucial to the success of software development projects. It is also a challenging and volatile process in practice. There is growing interest in enhancing the development of this shared understanding by improving related processes and support tools. The design and evaluation of such process improvements and tools should be based on robust theory and a clear understanding of the phenomenon and its context in practice. There is however, minimal empirical research on understanding the phenomenon of shared domain understanding in practice in this situation. This paper seeks deeper insights into the process of sharing domain understanding in the context of Small to Medium-sized Enterprises (SMEs) in the software development industry by investigating the barriers and challenges in practice. The study focuses on SMEs because of their economic importance globally and in New Zealand in particular. Small organizations may be especially challenged due to reliance on key individuals and insufficient resource to employ several domain specialists. In this paper we present the results of a field study of commercial software development practice in which we conducted semi-structured interviews with practitioners from ten such organizations. The study provides insights into practices and perceptions related to the challenges software development practitioners face in developing shared domain understanding with the client stakeholders. Our results identify a diverse range of challenges and barriers on which to relate theory and as a foundation for designing process and tool improvement.

**Keywords:** evidence-based software engineering, knowledge sharing.

## 1 Introduction and Motivation

The success of a software system is largely based on the degree to which it meets the expectations of a client stakeholder group in terms of addressing some real world problem. The software development process involves the software production team and client stakeholders developing a shared understanding the of the problem domain sufficiently well to determine the most appropriate set of features and attributes for a new software system, as part of the solution. This evolution of shared understanding involves a "mesh" of communications and knowledge sharing interactions between the software production (vendor) team and the client stakeholder group. This includes the client stakeholder group (e.g. users, managers, and domain experts) and the software production group (e.g. analysts, designers and developers) sharing and seeking knowledge using a variety of communication channels and information artefacts. This knowledge is integrated into their existing knowledge or understanding and combined and aggregated to form both teams' understandings of the application domain. Some of this domain understanding needs to be "shared" in some sense to avoid misunderstandings, misaligned expectations and allow cooperative progress. The notion of shared understanding is discussed in more detail in [1, 2] and is taken to refer to the degree of cognitive overlap and mutual beliefs, expectations and perceptions in the following discussions.

Although this domain knowledge sharing is most visible at the elicitation effort early in a software project, this process of articulating, sharing, clarifying, reflecting and sharing understanding is iterative and incremental throughout a software project. As software development progresses, the domain problem is often better understood and the business goals, business requirements and user requirements are consequently refined. The software system goals, requirements and specifications are then also refined to align with this better understanding of the problem domain. Although domain understanding needs to be shared *within* the software development team (and among the client stakeholders), the focus of this paper is the interactions and knowledge sharing across the production team and client stakeholder group boundary.

At the individual level it involves developing an understanding of the application domain, refining that understanding to a level that is appropriate for the role of that individual, and applying it at the time of "need". Thus individuals in the production team and client group are co-dependent to some extent on each other's problem domain knowledge and capabilities to share and internalise this knowledge to suit the individuals' and teams' needs. That evolving individual understanding needs to be periodically shared, "tested", verified and agreed upon so that those involved can work cooperatively towards the same goals that create sufficient value for the clients (and the clients understand this value!).

The sharing of domain understanding is challenging in practice because of the inherent complexities. It may involve a large number of individuals with a broad diversity of existing specialized capabilities, expertise and vocabularies, as well as different cultures, beliefs and values. Their motivations and authority to influence the development project can also be widely divergent. Furthermore the individual and team-level exchanges are characterised by cognitive, social and organisational interactions that are unpredictable and potentially error-prone. This includes, for example, challenging activities such as developing a shared vocabulary, sharing and internalising both conceptually abstract *and* detailed information about the problem domain, reconciling many points of view from diverse stakeholders and accommodating changing and volatile understanding. It also involves periodic verification of some shared representation of the domain understanding and how it maps to the software solution. To add to this complexity, a sufficient level of shared understanding between the two groups can never be certain because of the difficulty in not only defining what is "sufficient" but also in measuring the level of shared understanding. Instead, the misunderstandings, conflict and breakdowns that result from a lack of shared understanding are focused on. This paper takes the approach of identifying the barriers and challenges of sharing understanding, with a view to overcoming them with better tools or techniques.

Although it is inherently challenging, the development of shared understanding is critical to the success of a software project. Successful development of a software system is predicated on the vendor team's understanding of the main concepts, goals and purpose of the software system and how well this aligns with a client group's expectations. A number of researchers in RE (see for example [3-6]) argue convincingly for the central role domain knowledge sharing plays in RE activities. Empirical evidence from their studies of RE practice demonstrates that high quality requirements are crucially dependent on the client and vendor stakeholders sharing a sufficient level of understanding of the problem domain.

The need to understand and improve practice in this area is not lessening either, despite significant advances in modeling, tools and processes over the last few decades. We are seeing the application of software systems to an ever widening diversity of application domain, often conceptually challenging and complex. This broadens and deepens the domain knowledge that developers and other non domain experts have to understand. The need for further research into supporting and comprehending the phenomenon of "developing shared understanding" in practice becomes even more evident in the context of new types of software (e.g. ubiquitous, service oriented, self-managing, or mesh) and emerging development contexts (e.g. global development teams, distributed users, product or market driven development).

There has been considerable research into developing tools, techniques and processes to support the activities and complex interactions that contribute to developing a shared domain understanding in the context of emerging software needs. There seems to be "race" in tool and technique design between knowledge and implementation, however, as pointed out in [7] in the context of Human Computer Interaction design. They argue that "it is a typical pattern in HCI for new ideas to be first codified in exemplary artifacts and only later abstracted into explanations and principles". This pattern also seems to apply to the development of tools and techniques for other aspect of software engineering support. This may be a symptom of the relative immaturity of software development as an engineering discipline. New ideas may be implemented based largely on implicit or tacit knowledge, more of a characteristic of a "craft". As the tacit knowledge is made explicit, and forms the basis of design and implementation of new software development tools and techniques, the discipline is seen as moving more towards "engineering". With this in mind, this study seeks to gain a deeper, explicit knowledge of the challenges of the "phenomenon-in-action" of sharing domain understanding. This should then better support the selection of related theory from disciplines such as cognitive sciences, semiotics, knowledge management and organisational theory. This in turn will help to explain and diagnose the challenges and inform future tool and process design to address them explicitly. It also provides a clear theoretical basis with which to *evaluate* the tools or processes against. Then the purpose of evaluating a new tool or process would be to validate a distributed cognition model or knowledge transfer hypothesis, for example, rather than just understanding the tool. This study is the first part of a larger research project that follows this approach.

In addition, many of the findings and proposed approaches found in existing literature are aimed at large organisations, with the tacit assumption that these findings will apply to small organisation (i.e. having less than 50 employees [8]). This point is highlighted in [9], where the authors argue that RE in small organisations is under-represented in research literature. They further observe that such organisations make up a large part of the software industry) and in [10] they estimate that SMEs contribute 80% to economic growth worldwide. Moreover, it is likely that small organisations are more vulnerable to the complexities and volatility of developing a shared domain understanding compared to large organisations. It is widely acknowledged in literature that there are some fundamental operational differences between small and large organisations (see for example the Sept/Oct 2000 issue of *IEEE Software*). In empirical studies of small and medium organisations, [11], [12], and [13] characterise them as having fewer resources to devote to tools and hiring domain experts. Compared to their larger counterparts, small organisations appear to be more concerned about practice rather than “compliance” to formal, defined processes. They also observe that small organisations generally focus on shorter term priorities, which are typically directed towards deliverables. These ideas, strengthened by personal observations of small software companies, suggest that current understanding of and approaches to the challenges and barriers to shared domain understanding may not directly transfer to smaller organisations.

This paper addresses this lack of empirical information on challenges and barriers in small organisations in the area of sharing domain understanding, and examines the applicability of previously reported findings, generally drawn from experiences with larger organisations, to small companies. In addition, it is the intention of this research to gain insights into practitioners’ *perceptions* of their practices and challenges in this area. This is based on a desire to “know” the practitioners more deeply as “customers” of research and understand their experiences and needs in this area.

Having provided a brief background to the topic of shared domain understanding in software development, and justified the focus on empirical research in an SME context, the remainder of the paper describes and justifies the design and implementation of the field study (section 2) and describes and discusses the main findings in section 3.

## 2 Research Design and Implementation

The selection of a research methodology and specific data collection and analysis methods are based on the nature of the research aims and questions. This section describes and justifies these aspects of the research design and discusses the participating organisations and other features of the implementation of the study.

### 2.1 Aims and Methodology

It is the aim of this research to gain insights into challenges in the development of shared domain understanding in practice through practitioners’ perceptions. The scope of the field study undertaken includes exploring how practitioners conceptualise sharing domain understanding with client stakeholders, the techniques, tools and representations they utilize and their efficacy, and the importance they give to developing this shared understanding. The emphasis of this paper is on understanding the identified barriers and challenges, their causes and consequences, and approaches to addressing these challenges in practice.

In line with other exploratory studies of this type, a multiple case study method, with semi-structured interviews for data collection, is employed. A semi-structured interview was employed, rather than a formal, structured interview or survey, because it has the advantage of being able to clarify and probe issues and extend the focus of the discussion to interesting aspects *as they arise*. Thus, as observed by [14], a deeper and richer understanding of the phenomenon may be gained. In addition this technique encourages the development of a rapport and trust between the investigator and the interviewee. This is desirable if interviewees are to feel they can freely discuss their practices, challenges and successes.

Note that it is not the intention of this initial study to observe practices or analyse artefacts, which are also common sources of data in case studies. In addition, the viewpoint of the study is restricted to the perceptions of the software vendors, as represented by senior member of the participating software production teams. Comparison of the viewpoints of representatives of the client stakeholder groups is planned for a future study. This paper focuses on presenting the barriers and challenges to shared understanding and discussing the implications for researchers and practitioners.

### 2.2 Case Organisations

Invitations to participate in this study were sent to 204 organisations, based on the company size (small), and involvement primarily in software development. We selected candidates who had been operating for at least 5 years to allow for maturing of its practices.

Of the candidate organisations invited, 11 organisations initially agreed to participate and 10 organisations actually proceeded with the interviews. Experienced senior-level staff from the organisations were interviewed, with the view that they would have a clear overview of processes as well as some depth of interaction with client representatives, which proved to be the case. Two of the authors of this paper were involved in interviewing all the participants, one facilitating the interview, and the other taking detailed interview notes. The interviews were all recorded on audio tape for later transcription. The interviews were generally located at the place of work, or a neutral venue if requested, and lasted between one and two hours.

The rich and extensive set of data collected from the interviews includes information related to what the important challenges are to developing a sufficient shared understanding with the client stakeholders in practice and why. Thematic analysis of this data was employed as the method of data analysis. As noted in [15] this is a common method of analysing qualitative interview data in order to identify concepts or themes related to a phenomenon. The main construct being analysed is the *process* of developing shared domain understanding. As [16] points out, analysing a process may provide a holistic view of a system of action, which includes activities, roles, artefacts tools and techniques. The unit of analysis is the (small) organisation rather than specific teams or projects.

The participating organisations represent a wide diversity of application domains and include 3 product-driven companies and 7 providing bespoke software development services. All of the organisations had been operating for over 8 years and most of them closer to 20 years. The organisations would be classified as small enterprises with all having fewer than 100 Fulltime Equivalent (FTE) employees The representatives from the companies were all at a senior level ranging from senior systems analyst to company owners, with 8 of them having more than 10 years' experience in the software industry and the other two having extensive business or domain experience.

## 3 Challenges and Barriers to Shared Domain Understanding

This section presents the results of the investigation, discusses implications for practitioners, and speculates on some possible directions for addressing some of the issues.

All organisations had stories of miscommunications and misunderstandings that had contributed to project difficulties and acknowledged a number of challenges and barriers in developing a shared domain understanding. Table 1 depicts the main challenges identified in order of decreasing strength from this study. It should be emphasised that all the barriers are interrelated both causally and hierarchically and are represented in Table 1 according to the strength perceptions of the participants of this study. The relative strength of an identified challenge is estimated on a 10 point scale. It is based on the frequency with which interviewees identified it, the degree of prompting required, and the emphasis they placed on it (voice, body language, anecdotes). The rest of this section discusses the individual barriers presented in Table 2 in more detail, including any root causes and strategies for addressing the challenges identified by the participants in this study. Alignment with results and concepts from existing literature is also discussed.

**Table 1** Challenges and Barriers to Shared Domain Understanding

| Challenges and Barriers | Relative Strength |
|---|---|
| Inadequate client representation | 10 |
| Inter-group diversity | 8 |
| Lack of a common vocabulary | 8 |
| Lack of access to key stakeholders | 8 |
| Changes in problem understanding | 7 |
| Client uncertainty or disagreement | 6 |
| Difficult representations of understanding | 5 |
| Poor communications practice | 3 |

### 3.1 Client representation

The challenges presented by poor quality client representatives is consistently emphasised as a significant barrier to shared understanding by all participants. The prevalent view of the interviewees is that the client representative(s) is a key “interface” to the client organisation and if this relationship and interactions are poor then communications suffers and it is difficult to elicit domain knowledge and verify shared understanding. It seemed that this is generally a many-to-one (or few) relationship for the participating organisations. This can be expected to magnify the challenge of a poor representative compared to organisations with multiple points of contact, where a poor quality representative may be outweighed by other high quality representatives.

Characteristics of poor quality client representatives generally related to their perceived lack of domain knowledge, lack of availability or some form of "resistance" to sharing. Table 2 summarises the main challenges identified by the participants, in decreasing order of strength. These are now discussed in more detail including the perceived causes and practical strategies from both the field study and related studies from literature.

**Table 2.** Challenges Related to Client Representation

| Challenge/Barrier Identified |
|---|
| Lack of domain knowledge . |
| Lack of availability |
| Actively or passively resistant |
| Has a hidden agenda. |
| Poor learner |
| Indecisive |
| Lack of external authority |
| Overly demanding |

The most strongly emphasised characteristics of a poor quality client representation relate to perceptions of their lack of domain knowledge. This may be a deficiency in depth or breadth of domain knowledge. It was seen as resulting in an information need not being satisfied (shared) or sharing conceptual understanding being limited. It was also described as resulting in unnecessary uncertainty or volatility in understanding and requirements. Interestingly, "poor domain understanding" was also used to describe client representatives who were poor at articulating their (tacit) knowledge explicitly, or unable to communicate their understanding using terminology and concepts that the software production team could easily "digest". Also included in this category of "poor domain understanding" were representatives with little understanding of the knowledge or needs of the wider client stakeholder group.

Although it was a strongly identified barrier in this study, the interviewees did not offer any specific explanation regarding why client stakeholders with insufficient domain knowledge may be given the role of client representative. Some clues are found in literature, however. It is suggested in [4] that client representatives may be chosen because of their (high) position rather than domain knowledge. In addition, it is observed in [17] that in small organisations often the most capable (knowledgeable) client stakeholders are too busy contributing to business as usual to be client representatives. Another field study reported in [18] also notes the challenge of client representation with inadequate domain knowledge and concludes that a contributing factor is the lack of involvement of the development team in selecting the client representative(s). Indeed, several participants from our study suggested that this challenge could be alleviated by having more production team control of an identified stakeholder selection process, although it was remarked that clients would be unlikely to "allow" this. Production teams facing this barrier would generally rely on their own in-house domain expertise and would try to share their understanding with the client representative. Participants point out that this is a fairly limited solution, however, since in-house expertise is not always available, client representatives are not always open to "being advised" about their own application domain, and the specific contextual factors related to that particular client organisation, situated in a wider domain ontology, may be missed. It is certainly worth noting that this still seems to be an important challenge, over a decade after those previously mentioned related studies identified it as so.

Lack of availability of the client representative is also highlighted as being a significant barrier to sharing domain understanding. The discussion on this mirrors that for the challenge of accessing key stakeholders, discussed in detail in section 3.4 and so the reader is referred to that section for more detail.

Another area of inadequate client representation identified as noteworthy relates to problematic attitudes and behaviors of the representatives. Broadly speaking this is perceived as overt uncooperative behavior such as "holding back" feedback or information, or more passive resistance such as always agreeing, with little depth of thought. Active resistance to sharing understanding included refusing to do something requested by the production team. An unwillingness to compromise or negotiate and repeated disagreement on domain understanding were also seen as active resistance. Some participants interpreted a lack of accessibility of the client representative, or overly "secretive" client behavior (inappropriately commercially sensitive), as active resistance also. Passive behaviors such as verifying production team understanding with no challenge or discussion were viewed as less likely to lead to negative conflict, but still regarded as a barrier to shared domain understanding. Similarly, passive attitudes such as a general lack of engagement, or an unwillingness to commit to a position were also viewed as barriers. These concepts identified by the interviewees are close to the notions of "silent resistance" and "compliance resistance" noted in [19] as barriers to shared understanding in requirements engineering.

Participants described such uncooperative behavior as often hindering knowledge elicitation and verification activities throughout the software development lifecycle. They describe the consequences as potentially leading to: delays in the project, misaligned expectations, extra communications effort, lack of buy-in, lack of trust or heightened project risk.

The client representative's resistance to verifying and sharing understanding was seen as due to a low value being placed on the role by the client representative. This resulted in their lack of "buy-in", "commitment" or "resentment" to that role. Some participants also noted that certain personality traits, such as selfishness (thinking only of their own gain), could also disrupt communications and sharing understanding. This aligns with the observation in [17] that client stakeholder personalities may be at the root of these attitudinal and behavioral issues. It is suggested in [19] that resistance to the change brought about by the introduction of a new software system may trigger attitudinal or behavior problems in client stakeholders, and this may apply to the client representative.

The interviewees had few suggestions to address this resistant behavior and poor attitudes. One participant suggested that selecting an open-minded, tolerant *production team* representative with good communication skills may assist in modifying the client representative's behavior. Another participant suggested that support from another client stakeholder with higher authority should be sought.

Power is another clear barrier related to effective communication with the client representative. Some are perceived as playing power games with hidden political agendas, so that aspects of understanding were withheld to the advantage of the client in some way. It also encompasses situations where the client representative introduces their own ideas or agenda, without socializing it first with the other appropriate client stakeholders (particularly their managers). Related to this is the frustration expressed by some participants when trying to negotiate understanding and perhaps reach a compromise or decide on alternative views, when the representative doesn't have the authority to speak for the organisation and must consult with a higher authority.

It is clear from the interviews that most participants placed significant reliance on getting quality domain expertise and quality feedback and verification of understanding from the client representatives. They generally perceived the selection of the client representative(s) as largely out of their control, although two organisations report influencing the selection of the customer representative through negotiation. Another three organisations reported employing their own domain experts, who act as "proxy" clients. Presumably the client organisations don't actively select a poor quality representative for a project, so the question remains as to how this situation arises? No clear causes are offered by the participants, although the "blame" was certainly placed with the client group by the (vendor) interviewees.

While the quality of client representation is discussed in RE literature ([20], [21]) it appears that for small organisations it is perceived as a particularly significant and frequent barrier to developing shared understanding. Perhaps this is because larger teams in larger organisations have multiple points of contact. This could be a fruitful area for process improvement and better tool support. How can a more visible "client management" process be designed that will promote the selection of the client representatives based on appropriate criteria and support them to engage and commit, despite having competing work pressures? Or how can the knowledge sharing role of the client representative be made less crucial for an SME? One approach may be to support client stakeholders (and members of the production team) to explicate their relevant domain knowledge in a rich representational format that is straightforward to produce, manipulate and verify in a collaborative and distributed mode.

### 3.2 Inter-group Diversity

"Diversity" between the client and vendor groups is also identified strongly as a barrier to shared understanding. This is described by the participants as being linked to the differences in individuals' characteristics. This includes: their experiences, depth of knowledge, abilities to conceptualise, values, risk tolerance, and priorities. This barrier is conceptualised as resulting in "difference trends" between the groups that develop over a series of inter-group interactions. This can lead to unexpected actions, increased conflict, misunderstandings, miscommunications or misinterpretations that disrupt clear communications and hinder the development of shared understanding. Particularly noted by participants are the differences in depth of knowledge between the two groups, more technical on the software production team side and more business oriented in the case of the client group.

Three particular issues related to inter-group diversity were particularly emphasised by participants. One relates to the "difficulty to get them to speak the same language". This is seen as a significant barrier to developers gaining a sufficient understanding of the business processes and goals, and being the root cause of "some projects failing badly". This is discussed in more detail in the section on the challenges of developing a common vocabulary (with which to create, share and verify understanding).

Another barrier identified related to diversity, is the problem of the software production team "jumping into the coding process before they understand the business goals and processes". This was seen as resulting in a "cycle of change, change,

change", which is problematic to accommodate and could result in delays. The flip-side of this situation was also reported as a barrier to sharing domain understanding. This is the situation where clients inappropriately, and in the early stages of knowledge sharing, specify aspects of the *solution* system with flawed justification or poor understanding of the solution domain. Participants pointed out that this could restrict problem domain exploration and constrain consideration of alternative candidate domain solutions.

Another interesting challenge perceived by the participants relates to the lack of "big-picture" some client stakeholders exhibit. This is seen as resulting in stakeholders often getting "lost" and "missing the point because they don't understand what the business is trying to do". This relates to the notion of "Situation Awareness" (SA) defined in [22], and discussed in terms of inter-team communications in [23]. SA can be described as comprehension of the "big picture" of the elements of a situation. Cognitive psychology suggests that a level of SA is needed for sense-making and decision making regarding the situation, and how it might be in the near future. The high workload of client stakeholders (and possibly production team members) could impair their SA and be an impediment to internalising and sharing understanding. This could result in decision making based on the wrong understanding. This suggests that it may be useful to have some mechanism for developing and sharing the "big picture", in terms of business goals and aims, for example, and having this front of mind at key decision points in developing shared understanding.

It is worth noting that interviewees did not mention cultural diversity as a barrier to sharing understanding, although it is identified in literature as a common barrier to communications and shared knowledge (e.g. [24, 25]). It may be that the participating organisations may only serve a local market with little cultural diversity.

### 3.3 Lack of a Common Vocabulary

Although only identified by two organisations during the open questioning phase of the interviews, when prompted all but one participant agreed that a lack of a common vocabulary can be a barrier to sharing domain understanding. They described the issue as the use of "jargon", or "unfamiliar language" when sharing information for knowledge transfer or validation. Generally this was resolved by repeated clarification and verification interactions. They pointed out that this could be time consuming and cause delays, however. Also, sometimes individuals "didn't know that they didn't know" regarding the terminology and by the time they did recognize the confusion the impact on the project was more serious. This issue is also reported in literature and typically the maintenance of a shared glossary is recommended to alleviate the problem [3, 5]. Participants noted that this wasn't always useful because it the glossary was not always referred to by some client stakeholders (who maybe had no input to it). Other shortcomings given include: the glossary wasn't clear enough, it had gaps, or it wasn't up-to-date. Certainly it is restricted in the knowledge represented and provides limited information on concepts and their relationships. Perhaps a shared cognitive map, collaboratively developed and maintained, would serve this purpose better. Of course the overhead in developing and maintaining such a representation of the project "ontology" would need to be minimized in practice.

### 3.4 Lack of Access to Key Stakeholders

A variety of circumstances are identified as being the root causes of this access challenge including: geographical distance, delegation of responsibility, multiple layers of stakeholders, office policy, high cost of stakeholder involvement, or stakeholder indifference. There is a strong perception that stakeholders (including the client representative) are often too busy with other work (over-worked perhaps). Consequently they can take too long to respond to requests for information or confirmation of understanding, or provide shallow or incorrect responses due to this work pressure. Sometimes this was interpreted as the stakeholder having a lack of commitment or indifference to the project, because they don’t “make themselves available”.

One of the issues identified by participants with geographically distant client stakeholders is the lack of opportunity for face-to-face interactions and the concomitant rich communications available through this channel. Participants described the use of video conferencing as a partial solution to this. The use of a variety of communication channels (e.g. phone, email, SMS), was a common strategy described. This depended on the nature of the understanding to be shared (e.g. urgency, impact) and the disposition of the particular stakeholders involved, as well as the pragmatics of the situation.

Deficiency in availability often resulted in a lack of scheduled meetings or delays in the schedule, with consequent delays in sharing understanding and project progress. Participants also discussed the need for unscheduled access to certain client stakeholders, particularly the client representative. This was typically for clarification or verification of understanding where the need is urgent and possibly of less significance to the project. This often involved a quick phone call or email. Sometimes, if the required client stakeholder(s) was not available in a timely fashion, members of the production team would act on assumptions, decisions or interpretations that had not been verified with the client stakeholder group, increasing risk and limiting the sharing of understanding.

Multiple layers of stakeholders, for example where a client representative is a third party agent acting on behalf of the client, was also perceived as a challenge. The “thicker” layer of interpretation and possible increased communications times were seen as factors that could increase the chance of miscommunication or delays. One participating organisation report that one particular client had a policy that prevented the software development team from directly communicating with end-users. Although business analysts from the production team could communicate with the end-users, this potentially created a situation similar to the multi-layer stakeholders.

It may be that a persistent representation of some aspects of shared understanding that can be easily manipulated and annotated could provide a partial solution to this barrier by providing an asynchronous, distributed and ideally rich mechanism for sharing understanding. This could lower the accessibility pressure for some key stakeholders.

### 3.5 Changes in Problem Understanding

A commonly cited issue with requirements management is the difficulty in managing changes to scope and requirements [19]. Although not identified initially by participants, when prompted about changes to domain understanding, they expressed the view that, such change can be problematic if not "well managed". They described challenging experiences such as overly frequent changes, clients not sharing changes with the vendor, and lack of clarity on the wider impact of new understanding. Overall they saw the changes in understanding as positive and a natural part of the evolution of shared understanding.

### 3.6 Client Uncertainty or Disagreement

Over half of the participants indicated that client uncertainty with the problem domain is a barrier to sharing domain understanding. This is subtly different to insufficient domain knowledge. The client stakeholders may have sufficient depth of understanding but they are uncertain about which aspects of their domain knowledge apply or are important. A factor in this uncertainty may be that different stakeholders "compete" for their views to be the prevalent shared understanding.

Uncertainty with the envisioned the system goals was also seen as a possible barrier. While this is expected at the project concept stage and early phase requirements, some participants observed that this uncertainty could be a repeated pattern of and hamper decisions about shared understanding.

Another challenge to sharing domain understanding was described as the situation where there is a low overlap of some areas of understanding within the client stakeholder group. This may manifest as inconsistent, conflicting or competing points-of-view being "shared" by the client group. Participants observed that the client stakeholders may not be aware of this situation because they have had no need (or opportunity) to integrate or share these aspects of their understanding with each other before. This barrier is well acknowledged in literature and has been identified in other empirical studies such as [20, 26].

Most participants suggested that this challenge could be addressed by finding a mechanism to facilitate the client stakeholders to agree on important aspects of the problem domain, and "achieve buy-in" to these shared views, *before* they share this understanding with the software production team. Participants acknowledged that this is generally a challenge in practice. In [20] it is suggested that when the software team identifies the divergent client views they should organize a meeting with all the related (conflicting) client stakeholders together and facilitate a shared view. While this is not uncommon during early requirements elicitation, it is often difficult to get such a group together again in one place at one time because of divergent work schedules. A distributed, virtual mechanism of knowledge sharing that improves the visibility of all relevant clients stakeholders' points of view may assist with overcoming this barrier. It would have to be low effort and low complexity to be practicable, however.

### 3.7 Difficult Representations of Understanding

Participants observed that often the sharing of domain understanding involved the development and sharing of representations of aspects of the knowledge to be shared and verified. The representations identified by interviewees as representing shared domain knowledge includes formal and structured representations as well as: flow charts, business process diagrams, scenarios, use-cases, requirements specification documents, various UML diagrams, conversations and email threads. Screen shots, prototypes and product demonstrations were also identified as domain knowledge representations, with the reasoning that contain "embedded" (but constrained) understanding of the problem domain, and that they uncovered misunderstandings.

Participants perceive the use unfamiliar or overly complex representations as a barrier to sharing understanding. It is effectively introducing a new vocabulary that is not common to the two groups, as previously described. Interviewees emphasised the importance of being aware of the clients' fluency in interpreting and manipulating different knowledge representations and notations. This is in line with the findings of [14, 17, 26] who link the use of certain knowledge representations (UML, object oriented representations and use cases) with barriers to knowledge sharing. These studies also suggest that pictorial and concrete representations of knowledge (e.g. screen shots or prototypes) are typically easier for the client stakeholders to understand. This is supported by the experience of the participating organisations.

Natural language (both semi-structured and unstructured) was identified as the most common representation of shared understanding because it is the "lowest common denominator" for understandability. It was also noted that the ambiguity inherent in natural language is a source of challenge to sharing understanding and difficult to manage, as also reported in [14, 27] in relation to requirements specifications. The mechanism for discovery of ambiguity in natural language knowledge representations typically involved reviews, cross validation against other representations and frequent clarification interactions. One participant had a strong conviction that written communications "in any form" are always ambiguous and would never be a substitute for regular face-to-face meetings where more signs are available to identify ambiguity and instant clarification is possible. [11] suggests that being able to detect ambiguity and imprecision in natural language is a skill that that can and should be learned.

A few participants noted that poor presentation of knowledge representations can be a barrier to shared understanding. They described "unattractive", "dry". "dull", and "voluminous" documents for review or confirmation of understanding as de-motivating and limiting client engagement with the knowledge sharing activity.

### 3.8 Poor Communications Practice

Communications and sharing understanding are closely related [4] and so poor inter-group communications are strongly linked to challenges in developing shared understanding. The communications barriers identified by interviewees generally related to deficiencies in the timeliness or frequency of the communications, or a lack of "rich" communications interactions. These challenges were seen as arising from insufficient communications planning, lack of stakeholder availability (for all the reasons previously discussed), unrealistic project timeframes, or the use of project processes or methodologies that de-emphasised client communications. Poorly defined or socialised roles and responsibilities are also perceived as a source of challenge to communications practice. This is one of the factors identified in [4] also.

Clearly sharing understanding benefits from the availability of a diversity of communications channels, knowledge artefacts and communication related roles. Multi-view models of communications interactions, such as that introduced by [28] may provide fuller understanding of the effectiveness of communications for sharing understanding.

## 4 Conclusion

Overall, participants identified a broad range of interrelated barriers and challenges to adequate sharing of domain understanding. The potential for client representatives to inhibit the sharing of domain understanding between the two groups was emphasised in the frequency and strength with which interviewees, unprompted, raised this as an issue. The next most forcefully expressed barrier related to the diversity in "world views" and experiences of the vendor and client groups. This contributed to a number of communications issues that obstructed sharing understanding between the two groups.

This study achieved its aim of deepening the understanding of the barriers and challenges faced by software development teams in collaboratively sharing domain knowledge with the client stakeholder group. This will now be used to guide the development of multi-view models of sharing domain understanding in this context. Theories from cognitive science, organisational theory, semiotics and knowledge management will be considered for candidate principles to inform the models. The longer term plan is to then test the models through the design and evaluation of tools, techniques and processes based on these models.

## References

[1] J. A. Cannon-Bowers and E. Salas, "Reflection on shared cognition.," *Journal of Organizational Behavior,* vol. 22, pp. 195-202, 2001.

[2] S. G. Cohen and C. B. Gibson, "In the beginning: Introduction and framework," in *Virtual teams that work: Creating conditions for virtual team effectiveness*, C. B. Gibson and S. G. Cohen, Eds. San Fransisco: John Wiley & Sons, 2003.

[3] E. G. Alcázar and A. Monzón, "A process framework for requirements analysis specification," in *Proceedings of 4th International Conference on Requirements Engineering*, 2000, pp. 27-35.

[4] J. Coughlan, M. Lycett, and R. D. Macredie, "Communication issues in requirements elicitation: a content analysis of stakeholder experiences," *Information and Software Technology,* vol. 45, pp. 525-537, 2003.

[5] R. Offen, "Domain Understanding is the Key to Successful System Development," *Requirements Engineering,* vol. 7, pp. 172-175, 2002.

[6] A. Osada, D. Ozawa, H. Kaiya, and K. Kaijiri, "The role of domain knowledge representation in requirements elicitation," in *25th IASTED International Multi-Conference Software Engineering*, Innsbruck, Austria, 2007, pp. 84-92.

[7] A. G. Sutcliffe and J. M. Carroll, "Designing claims for reuse in interactive systems design," *International Journal of Human-Computer Studies,* vol. 50, pp. 213-241, 1999.

[8] EU, "The new SME definition. User guide and model declaration. ," European Commission, 2005, http://www.ec.europa.eu/enterprise/enterprise_policy/sme_definition/sme_user_guide.pdf

[9] J. Aranda, S. Easterbrook, and G. Wilson, "Requirements in the wild: How small companies do it," in *Proc. 15th IEEE International Requirements Engineering Conference RE '07*, 2007, pp. 39--48.

[10] S. Pavic, S. C. L. Koh, M. Simpson, and J. Padmore, "Could e-business create a competitive advantage in UK SMEs?," *Benchmarking: An International Journa,* vol. 14, pp. 320-351, 2007.

[11] E. Kamsties, K. Hörmann, and M. Schlich, "Requirements engineering in small and medium enterprises," *Requirements Engineering,* vol. 3, pp. 84-90, 1998.

[12] F. Pino, F. García, and M. Piattini, "Software process improvement in small and medium software enterprises: a systematic review," *Software Quality Journal,* vol. 16, pp. 237-261, 2008.

[13] A. Atherton, "The uncertainty of knowing: An analysis of the nature of knowledge in a small business context," *Human Relations,* vol. 56, pp. 1379-1398, 2003.

[14] L. Karlsson, A. G. Dahlstedt, B. Regnell, J. N. Dag, and A. Persson, " Requirements engineering challenges in market-driven software development - An interview study with practitioners.," *Information and Software Technology,* vol. 49, pp. 588-604, 2007.

[15] M. Miles and A. Huberman, *Qualitative Data Analysis*. Sage, CA: Thousand Oaks 1994.

[16] W. Tellis, "Application of a case study methodology," *The Qualitative Report,* vol. 3, 1997.

[17] K. E. Emam and N. H. Madhavji, "A field study of requirements engineering practices in information systems development," in *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, 1995, pp. 68-80.

[18] A. Al-Rawas and S. Easterbrook, "Communication problems in requirements engineering: A field study," in *1st Westminster Conference on Professional Awareness in Software Engineering,*, London, 1996.

[19] H. Saiedian and R. Dale, "Requirements engineering: Making the connection between the software developer and customer," *Information and Software Technology,* vol. 42, pp. 419-428, 2000.

[20] L. Cao, & Ramesh, B., "Agile requirements engineering practices: An empirical study," *IEEE Software,* vol. 25, pp. 60-67, 2008.

[21] C. Lu, Chu, W. C., C. Chang, and C. Wang, "A model-based object-oriented approach to requirements engineering (MORE)," in *31st Annual International Computer Software and Applications Conference*, 2007, pp. 153-156.

[22] M. R. Endsley, "Toward a theory of situation awareness in dynamic systems," *Human Factors* vol. 37, pp. 32-64, 1995.

[23] C. A. Bolstad, P. Foltz, M. Franzke, H. M. Cuevas, M. Rosenstein, and A. M. Costello, "Predicting situation awareness from team communications," in *51st Annual Meeting of the Human Factors and Ergonomics Society*, Santa Monica, CA, 2007.

[24] D. Damian, "Stakeholders in global requirements engineering: Lessons learned from practice," *IEEE Software,* vol. 24, pp. 21-27, 2007.

[25] Y. Hsieh, "Culture and shared understanding in distributed requirements engineering," in *International Conference on Global Software Engineering*, 2006, pp. 101-108.

[26] M. Lubars, Potts, C., & Richter, C. (1993). A review of the state of the practice in requirements modelling. Proceedings of IEEE International Symposium on Requirements Engineering, 1993, 2-14.

[27] M. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Transactions on Software Engineering and Methodology,* vol. 14, pp. 277-330, 2005.

[28] J. Aranda and S. M. Easterbrook, "Distributed Cognition in Software Engineering Research: Can it be made to work," in *First workshop on Supporting the Social Side of Large Scale Software Development (SSSLSSD)* Banff, Alberta, 2006.

# Differences of Time between Modification and Re-modification: An Analysis of a Bug Tracking System

Akinori Ihara, Masao Ohira, and Ken–ichi Matsumoto

Graduate School of Information Science, Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, JAPAN 630-0192
{akinori-i,masao,matumoto}@is.naist.jp

**Abstract.** Managers of open source projects need to understand time to resolve bugs, which are reported into a bug tracking system on a daily basis, to make a release plan. Hewett et al. proposed an empirical approach to predicting time required to repair bugs. However, the predictive model did not distinguish between time to modify a firstly-reported bug and time to re-modify a bug. In this paper, toward predicting time to resolve bugs with accuracy, we identify such the differences of time between bug modifications and re-modifications. We have conducted a case study using Firefox project data. As a result of this case study, we have confirmed that time to resolve a firstly reported bug was shorter than time to re-modify a bug.

**Key words:** open source software development, bug tracking system, time to resolve bugs, bug modification process, Firefox

## 1 INTRODUCTION

As open source software with a large number of users increases, it is required to release a new feature or a bug fix on regular basis. Therefore, managers of OSS projects need to understand time to resolve bugs which are reported into a bug tracking system on a daily basis, in order to make a release plan. Hewett et al.[2] proposed an empirical approach to predicting time for repairing bugs. The study presented a bug modification process using a bug tracking system as a state transition diagram and predicted time spent for transition to each state. However, the predictive model did not distinguish between time to modify a firstly-reported bug and time to re-modify a bug. Time required for re-modifying a bug would be shorter than time required for modifying a firstly reported bug, since a problem in source codes must be more clear, compared to the modification of a firstly reported bug which may contain unknown problems. It will also take a longer time to resolve bugs, if developers in charge of bug modifications frequently change. In this paper, toward predicting time to resolve bugs with accuracy, we identify such the differences of time between bug modifications and re-modifications. We have conducted a case study using Firefox project data.

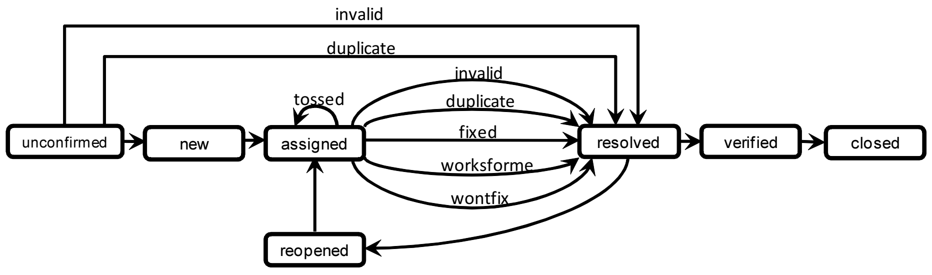


**Fig. 1.** A bug modification process[3]

## 2 RELATED WORK

There are many studies on bug modification processes with bug tracking systems in open source projects[1][3][4][5]. Focusing on time to resolve bugs in the bug modification process, we have proposed an analysis method to understand a factor which results in prolonging the bug modification process[4]. This analysis method represents a bug modification process as a state transition diagram and calculates the amount of time required to transit between states. We have conducted two case studies of the reported bugs in Apache and Mozilla projects. As a result of our analysis, we have found that the both projects needed long time to resolve bugs in the modification phase and verification phase. However, we could not achieve a clear understanding on the differences of time between bug modifications and re-modifications.

## 3 BUG MODIFICATION PROCESS

### 3.1 Bug Modification Process with a Bug Tracking System

Most open source projects use bug tracking systems to unify management of bugs found and reported by developers and users in their projects. A bug tracking system helps open source project's managers to know the progress of bug modifications, to avoid leaving unmodified bugs and so forth. Popular bug tracking systems include Bugzilla, Mantis, RedMine, Trac and so on.

Figure 1 represents a bug modification process using a bug tracking system. Although a bug modification process using a bug tracking system slightly differs among individual bug tracking systems, it substantially can be represented as a state transition diagram in Figure 1.

### 3.2 Modification Work

This section describes differences of the modification work flow between modification of a firstly reported bug and re-modification of a bug. Figure 1 shows the bug modification work flow. The modification work flow for a firstly reported bug is as follows. First, developers understand contents of bug reports. Second, developers understand a source code containing bugs. System names containing bugs are written in many bug reports. One developer often faces with the difficulty in modifying bugs by oneself, because most software systems work with other systems. Furthermore, developers need to consider the influence of their modifications on other source codes, due to such as the dependency of software modules. In contrast, re-modifying bugs would be finished by a shorter time, because analyzing the prior modification helps developers identify the reason and/or location of bugs. For the reasons mentioned above, we consider that time to resolve is affected by the presence or absence of the history of modifications.

## 4 ANALYSIS METHOD

This section describes a method for identifying the differences of time between bug modifications and re-modifications. At first, we describe a method to calculate time to modify a firstly reported bug and time to re-modify a bug. Time to modify a firstly reported bug is defined by the mean time from acceptance of the bug (**new**) to resolution of the bug (**resolved**). In contrast, time to re-modify a bug is defined by time from the decision of re-modification of the bug (**reopen**) to resolution of the bug (**resolved**). Some reported bugs are often needed to be re-modified several times. In this case, we calculate time for each re-modification of bugs. For example, if a bug are re-modified twice, we count time for the two re-modifications Therefore, time required to re-modify a bug depends on the number of changes of developers in charge [3]. In this paper, time to modify a firstly reported bug and time to re-modify a bug are respectively calculated by the number of modifications of each developer in charge.

## 5 CASE STUDY

### 5.1 Target Projects and Data

In the Mozilla Firefox project, Bugzilla is used to manage reported bugs. The Mozilla Firefox project has been developing a web browser product with a rapidly increasing share. The product is very popular due to the extensibility of functions (i.e., add-ons). The project has been using Bugzilla since 2001. In the case study, history data of Bugzilla in Firefox version 1.0, 2.0, and 3.0 had been examined. In this paper, we analyzed 10,917 bug reports. Their bug reports is closed bugs from 2003 to 2008. In the bug reports, the number of re-modified bugs were 434 and the number of firstly reported bugs were 969.

### 5.2 Result

Table 1 respectively shows time to modify a firstly-reported bug and time to re-modify a bug, by the number of changes of assigned developers. If an assigned developers never changed, the number of changes of assigned developers is shown as zero. Time to modify firstly modified bugs with zero, once and twice assigned developers is longer than time to re-modify bugs with zero, once and twice assigned developers. In addition, the number of firstly modified bugs excepting zero assigned developers is 426 of 969 (44%). The number of re-modified bugs excepting zero assigned developers is 107 of 434 (25%).

Next, we confirmed the significant difference of the number of modifications between the two kinds of the modification works, using Mann-Whitney U test[1]. In this case study, the significance level of a test is 5%. Table 2 shows the test result. In case of zero and once assigned developer, there is significantly different between the two kinds of the modification works. However, In case of over two times assigned developers, there is no significance different between two works.

**Table 1.** Time to firstly-modify and re-modify bugs

| | firstly-modified bugs | | | | | re-modified bugs | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| number of assigned developers | zero assigned | once | twice | third times | more than four times | zero assigned | once | twice | third times | more than four times |
| median(days) | 11.0 | 24.3 | 82.7 | 129.3 | 429.2 | 0.6 | 8.2 | 75.7 | 163.2 | 227.4 |
| average(days) | 78.2 | 113.2 | 197.3 | 363.5 | 331.2 | 28.6 | 94.8 | 298.8 | 201.8 | 286.6 |
| variance(days) | 167.4 | 227.5 | 348.0 | 439.3 | 212.1 | 98.8 | 170.9 | 627.9 | 160.6 | 210.1 |
| number of bugs | 543 | 353 | 47 | 19 | 7 | 386 | 74 | 20 | 8 | 5 |

**Table 2.** Mann-Whitney U test

| number of assigned developers | zero assigned | once | twice | third times | more than four times |
|---|---|---|---|---|---|
| p-value | 2.2e-16 | 0.02 | 0.72 | 0.40 | 0.83 |

## 6 DISCUSSIONS

Based on the results of our case study, this section discusses the necessity of analysis on the differences of the mean time to resolve bugs due to kinds of bug modification process toward building a predictive model of bug resolution time.

As a result of this study, we observed that time to modify firstly reported bugs was shorter than time to re-modify a bug in spite of the number of assigned

[1] The data with this study has element counts of the population and each data set is not normally-distributed. Mann-Whitney U test is used when two samples are self-dependence.

developers. We also found that the number of assigned developers in modifying firstly reported bugs was larger than that in re-modifying bugs. In addition, the number of assigned developers in re-modifications is less than that in modifications of firstly reported bugs. Therefore, we think that time to understand bug reports and source codes with bugs is less required in re-modifying bugs.

In this paper, we did not analyze time to resolve bugs, considering every developer's skills and priority and/or severity of bugs. If a developer has high skill, time to resolve bugs would be shorten in nature. Also, if high severity bugs are reported, developers would modify such the bugs by priority. In the future, we would analyze time to resolve bugs, considering such the skills and priority and/or severity of bugs.

## 7 CONCLUSION AND FUTURE WORK

In this paper, toward predicting time to resolve bugs with accuracy, we identify such the differences of time between bug modifications and re-modifications. We have conducted a case study using Firefox project data. As a result of this case study, we have confirmed that time to resolve a firstly reported bug was shorter than time to re-modify a bug.

We think that the verification work also different between time for a first bug modification and time for re-modification of a bug. Finally, we would like to build a predictive model of bug resolution time, analyzing differences of time to resolve bugs modification considering bug modification processes.

## 8 ACKNOWLEDGEMENT

This research is being conducted as a part of the Next Generation IT Program and Grant-in-aid for Young Scientists (B), 20700028, 2009 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

## References

1. I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 international working conference on Mining software repositories (MSR'08)*, pages 145–148, 2008.
2. Hewett. R, and Kijsanayothin. P. On modeling software defect repair time. *Empirical Softw. Engg.*, pages 165–186, 2009.
3. Jeong. G, Kim. S, and Zimmermann. T. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering on European Software Engineering Conference and Foundations of Software Engineering Symposium* , pages 135–144, 2009.

4. Ihara. A, Ohira. M, and Matsumoto. K. An analysis method for improving a bug modification process in open source software development. In *Proceedings of the Joint international and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (EVOL) Workshops* , pages 111–120, 2009.
5. 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*, 11(3):309–346, 2002.

# Usage Result of Problem Resolution Information Sharing System for a Software Engineering Course

Atsuo Hazeyama Kazuyuki Shimada and Yusuke Kobayashi

Department of Information Science, Tokyo Gakugei University
4-1-1 Nukuikita-machi, Koganei-shi, Tokyo, 184-8501 Japan
hazeyama@u-gakugei.ac.jp

Abstract. Software development is highly knowledge-intensive and collaborative work. Problem resolution processes are performed iteratively during software development. The authors have proposed a problem resolution process model that was based on reflection and collaboration for a software engineering project course. They have also developed a support system based on the process model and applied it to an actual university course for two years. The results from the stored log data and the contents from two years' usage showed that similar trend on the number of registered problem resolution information, ratio of classification by phase and by contents was seen in both years (problem resolution information on coding with respect to phase was registered most, and most information was that on technical with respect to contents). We observed knowledge transfer by the teaching staff in the discussion space and in the bulletin board system of group.

## 1. Introduction

Software development is knowledge intensive and collaborative work. Developers face many problems during software development and solve them by interactions with various resources [10]. There are no developers who possess all required knowledge. They gather necessary information while they progress development. They may ask others who possess expertise [10]. Ye described importance of knowledge collaboration in software development and developed a software engineering environment that enabled to register and retrieve sample programs, browse past discussions for the programs, and inquiry professionals [10].

Ishida et al. pointed out some issues on information and know-how sharing in industrial software organizations as follows: although software developers and/or managers have desire to know-how sharing, it does not function well [6].

Problem resolution information sharing systems have been developed for some areas [1, 2]. Answer Garden was developed to use in a help desk. It provided a branching network of diagnostic questions that helped users find answers. If the answer is not present, the system automatically sends the question to the appropriate expert and the answer is returned to the user and it is inserted into the branching network [1]. Although Answer Garden contributed to efficient problem resolution, it did not seem to focus on a learning aspect such as reflection.

Reflection is known as a process that is rooted knowledge acquired through problem resolution processes. Hatamura advocated "Shippaigaku" that learned from failures in order not to iterate similar ones [4]. "Shippaigaku" specified six attributes, i.e., event, background, progress, cause, disposition, and lessons learned, in order to describe a failure and transfer it to others.

We have been conducting a group-based software engineering project course. The goal of this course is for students to acquire knowledge and skills that are necessary for software development through their experience of collaborative software development by group. We proposed a problem resolution process model by collaboration and reflection, which aimed to solve problems that occurred during software development and to be rooted knowledge acquired through problem resolution [5]. We also developed a system based on the process model to share the problem resolution information (description of problem, the resolution process, solution, and lessons) students encountered. This system is different from an issue tracking system like Bugzilla, Dhruv [9, 2] from the viewpoint of the target scope. Issue tracking systems manage bug information and information on the disposition in the testing phase. On the other hand, we manage problem resolution information students encounter in all phases of software life cycle. We reported some results gained from its application to an actual university course as follows: 1) around 90% of the registered problem resolution information dealt with the implementation phase, 2) 95% of the registered problem resolution information dealt with the technical information and only 5% dealt with process information [5].

We would like to ascertain whether this is a transient phenomenon or not. We also ascertain state of reuse of shared knowledge.

The organization of this paper is as follows: section 2 gives an overview of our problem resolution process. Section 3 presents a support system. Section 4 reports application of the system to an actual university course and some results. Finally we conclude this paper.

## 2. A Process Model for Collaborative Problem Resolution

The problem resolution process we propose is consisted of the following four steps as shown Figure 1.

**(1) Identification of a problem**

When a learner or a group faces a problem, (s)he identifies the event and its background. Problems may come from results of inspection and/or testing, troubles in programming, and/or troubles in development environments.

**(2) Information gathering**

A learner or a learning group collects information which is necessary for problem resolution and considers causes of the problem. As the information source for problem resolution, we assume the supporting system we provide in this paper and/or external resources such as Web pages. We also assume to collect information by means of communications with peers in the course and/or the teaching staff.

**(3) Disposition of the problem**

The learner deals with the problem to remove the cause, which was considered based on the information collected in the step (2).

**(4) Reflection**

When the problem was solved, the learner reviews the problem resolution process and describes the lessons learned by the problem resolution process.

We ask learners for describing items that correspond to the steps. We adopt "Shippaigaku" as a framework for describing the problem resolution process as shown in section 1. Shippaigaku is a discipline, which aims at learning from failures and prevents similar failures by sharing them [4]. In the learning process, we regard reflection as activities of describing all the attributes specified by "Shippaigaku" for the corresponding problem. The problem resolution information that will be stored through the abovementioned learning process is available for other learners. At this time we think learning effectiveness will not be obtained if a learner only refers to the information. Therefore we provide a function that referrers describe their lessons based on the information they referred. These lessons lead to feedback information for the submitters of the problem resolution information.

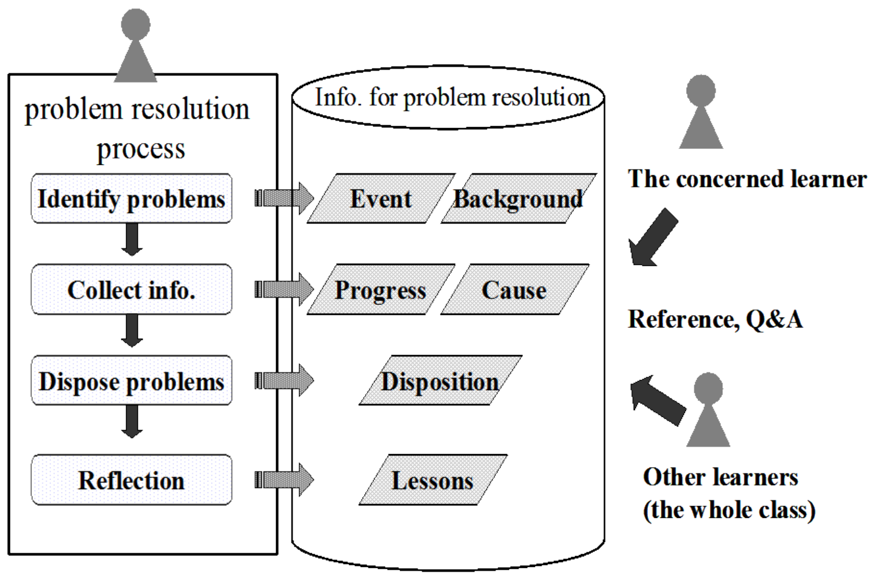
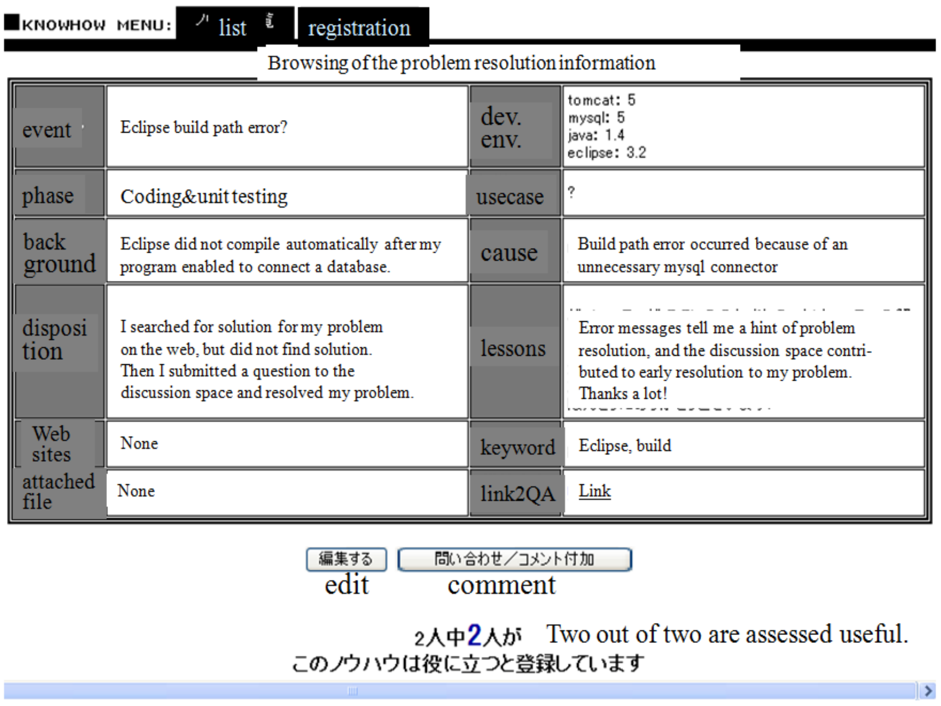


Fig. 1 Problem Resolution Process Model

## 3. Support System

We implemented a support system based on the abovementioned problem resolution process model. The system was implemented as a web application and a sub-system of "Shin-Gi-Tai" (Mind-Skill-Force) that was a group-based collaborative software development environment we developed [7]. "Shin-Gi-Tai" provides the following functions: document management, BBS (Bulletin Board System)-based communication support, project planning and progress reporting, and issue tracking. The system we developed in this study has the following major functions. The information that is stored by the functions is available to all groups in "Shin-Gi-Tai".

* Registration of the problem resolution information: a user registers the problem resolution information. By registering it according to the input form that corresponds to the attributes defined by "Shippaigaku", the context information is attached and reflection will be made by the user.
* Browsing of the problem resolution information: users browse the registered problem resolution information. They can evaluate the information and/or give comments for it. Figure 2 shows an example screen shot of the problem resolution information.
* Discussion space for the problem resolution: this function creates a space for discussions specific to an encountered problem. A user who would like to ask a question registers his/her question. In particular, as the context information is important for the encountered problem to be answered, description on the goal and on what (s)he has tried is mandatory. Then users enable to take three types of actions, i.e., browsing of questions, submission of messages, and browsing of the messages. From this function, a user can register the problem resolution information if the problem was resolved. The system associates the problem resolution information with the discussions.



| | | | |
|---|---|---|---|
| event | Eclipse build path error? | dev. env. | tomcat: 5<br>mysql: 5<br>java: 1.4<br>eclipse: 3.2 |
| phase | Coding&unit testing | usecase | ? |
| back ground | Eclipse did not compile automatically after my program enabled to connect a database. | cause | Build path error occurred because of an unnecessary mysql connector |
| disposi tion | I searched for solution for my problem on the web, but did not find solution. Then I submitted a question to the discussion space and resolved my problem. | lessons | Error messages tell me a hint of problem resolution, and the discussion space contri-buted to early resolution to my problem. Thanks a lot! |
| Web sites | None | keyword | Eclipse, build |
| attached file | None | link2QA | Link |

2人中2人が Two out of two are assessed useful.
このノウハウは役に立つと登録しています

Fig. 2 Screen shot of browsing the problem resolution information

Nakakoji et al. categorized communication in software development into coordination communication and expertise communication [8]. This function corresponds to expertise communication. Coordination communication is done in bulletin board system (BBS) of each group.

## 4. Evaluation

We show the results from two years' application of the system.

### 4.1 Overview of the course

We applied the system to an actual software engineering project course at our university in the 2007 and 2008 academic year. In this course, four or five students form a group. Each group selects one task from the two given by the instructor and is expected to complete their development via requirement analysis, design, implementation, and testing from scratch. Each group is required the system is implemented as a Web application with the Java technologies. During the project, the teaching assistants and the instructor (hereafter we call them the teaching staff) conduct inspection for requirement specification and several design documents. Faults detected during the inspection are required to be revised (the follow-up step was conducted). Acceptance testing is performed by the teaching staff for the system that is uploaded to a server machine and the development group conducts system testing for the same system. Faults are kept track by an issue tracking tool we developed.

Our department offers one and half years' programming courses by C language, one semester's course "automaton and language theory," and one semester's course "introduction of software engineering" just before the project course as related to the project course. "Introduction of software engineering" gives lectures on some software life cycle models, concept of object-orientation, modeling by UML (Unified Modeling Language), Web application development with Java technologies (including exercises). However, as all knowledge necessary for the assigned task can't be taught in the course, the students are required to investigate and/or exchange information to resolve their encountered problems in the project course.

In the 2007 academic year, the number of student developers was twenty-two and five groups were organized. The development was during 31 October 2007 through 22 January 2008. All groups finished their requirement analysis and design phases till mid December 2007. Integration of source codes written by their members was conducted around 10 January 2008. System testing by developers and acceptance testing by the teaching staff were done in the following two weeks.

In the 2008 academic year, the number of student developers was twenty-six and six groups were organized. The development was during 30 October 2008 through 21 January 2009. All groups finished their requirement analysis and design phases till mid December 2008. Integration of source codes written by their members was conducted around 10 January 2009. System testing by developers and acceptance testing by the teaching staff were done in the following two weeks. Table 1 summarizes a profile of the two years' projects.

Table 1 Summary of two years' projects.

| Items | Year | |
|---|---|---|
| | 2007 | 2008 |
| No. of students | 22 | 26 |
| No. of groups | 5 | 6 |
| Duration of project | 31 Oct. 2007 – 22 Jan. 2008 | 30 Oct. 2008 – 21 Jan. 2009 |

All groups finished their development in both years. At the start of the project, we asked all the students for registering at least one problem resolution information they encountered in the project. To register duplicated information with others is allowed if the information was acquired in the course of a student's problem resolution process.

## 4.2 Results

We show the results from the following viewpoints: classification by phase, classification by contents (technical versus process), and situation of knowledge collaboration.

### 4.2.1 Classification of the problem resolution information by phase

39 problem resolution information were registered in 2007 and 42 in 2008 (a student registered 1.8 in average in 2007, and 1.6 in average in 2008). Figure 3 shows the ratio of the registered problem resolution information by phase in the 2007 and 2008 course. We classified the phase into requirement analysis and specification, design, coding (including information on a development environment), testing, project management (PM), and others. As this figure shows, both courses present similar trend by this classification. That is, information on coding was registered quite most and then information on design was registered. No information on requirement analysis and PM was registered.

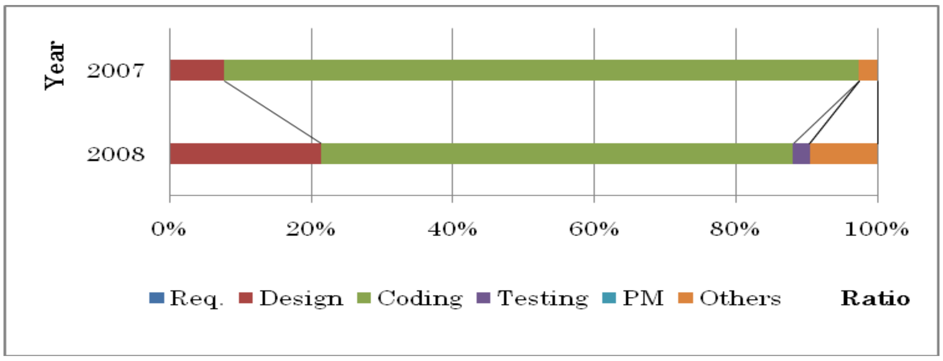


Fig. 3 Classification of the registered problem resolution information by phase

### 4.2.2 Classification of the problem resolution information by content

We classified the problem resolution information into categories proposed by Ishida et al. [6]. We classified the content into technical information and process information. Figure 4 shows the result. As this figure shows, more than 90% were technical information. Figure 2 shows an example of the registered problem resolution information.

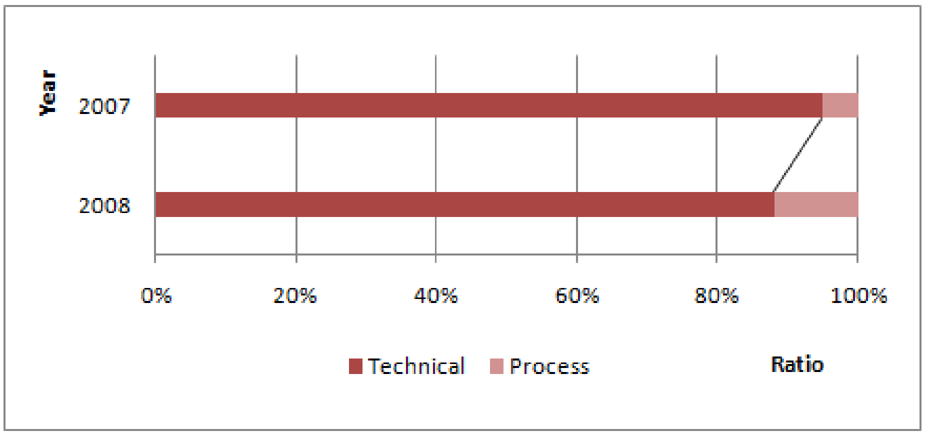


Fig. 4 Classification based on the contents of the registered problem resolution information.

### 4.2.3 Result on knowledge collaboration

#### (1) Knowledge collaboration in the discussion space

In the discussion space, eight questions were submitted in 2007, and one was submitted in 2008. They all led to resolution. Six questions were resolved by only one response. One question was followed by a counter question by a teaching assistant (TA) because the contents of the question were too short to answer it. In this case, the student who asked this question described detailed explanation, in addition, seemed to find a root cause for the problem as the result of his/her investigation. The TA who asked a counter question gave advice with respect to a solution to remove the cause. The rest of questions was “Current date-time can’t be inserted into a database” and “Build problem.” These two questions were resolved by exchanging messages between the questioner and responder(s) several times.

Question “Current date-time can’t be inserted into a database” was resolved by the following process: a teaching assistant asked the questioner for presenting details on the current situation and what (s)he wanted, and gave advice based on the responses from the questioner and the investigation results from internet search by the teaching assistant. Finally the problem was resolved in five turns.

Question “build problem” was also resolved by exchanging messages between a questioner and responder(s). In this case, first of all, a student (S1) asked the following question *“I created a file using Eclipse and updated it. After updating, I executed a program. However, the update was not reflected.” A* member of another group (P1) predicted a root cause. However, (s)he did not suggest a concrete solution. Then a teaching assistant (P2) asked the questioner for several inquiries and gave several advices, but the advices did not resolve the problem. Furthermore another teaching assistant (P3) who is a master course student gave advice to check a log file. By contents of the log file presented by the questioner, P2 found out a solution and gave a concrete advice to solve the problem. We show the message sequence as follows. This case worked escalation mechanism well.

Questioner (S1) at 21 December 2007: *I created a file using Eclipse and updated it. After updating, I executed a program. However, the update was not reflected.*
P1 at 08:37, 21 December 2007: *I think build is not done.*
Questioner (S1) at 12:46, 21 December 2007: *Maybe I think so. I check "automatic build" now. I checked "build all" and tried to build my program, but it did not work. Where should I modify?*
P2 at 14:31, 21 December 2007: *Do you mention your update is not reflected into the program? Cache may cause your problem. Are there errors in the "work" folder?*
Questioner (S1) at 14:43, 21 December 2007: *Where should I examine for that matter?*
P2 at 15:16, 21 December 2007: *There will be "work" folder under your eclipse project. Compiled JSP files are stored there. Cache will also be stored there. Troubles happen if the folder has errors or old cache remain. I have one question: do you have any concrete error messages? Isn't your update of files merely recognized by Tomcat?*
Questioner (S1) at 15:35, 21 December 2007: *Maybe I think so. I wrote "System.out.println();" in a servlet program, but it was not executed. I have deleted the contents of the "work" folder. Was that incorrect?*
P2 at 15:54, 21 December 2007: *No. If you find "X" mark on the "work" folder, please delete the contents of the "work" folder. In what situations did this problem emerge? That is, did you encounter this problem suddenly while the program ran well till yesterday? Or has not the problem resolved so far? Is the version of Java compiler 1.4? You can check the version and level of Java compiler by menu "window" -> "setting" -> "Java" -> "compiler" -> "compiler level" in eclipse.*
P3 at 16:34 21 December 2007: *If you can't resolve your problem by checking the abovementioned advice, please check the following: is a message "project can't be built until the build path is resolved" shown in a list of "problem" from "presentation of view" of "window" menu?*
Questioner (S1) at 18:04, 21 December 2007: *Dear P2, I suddenly encountered this problem. Yesterday, the system ran well. I checked the version number and level of the compiler. The level was 1.4.*
*Dear P3,*
*I found following two error messages: "The project will not be built until error with respect to the build path is resolved" and "a library necessary for project 'C:\eclipse3.2.1\workspace\KOKKO\WEB-INF\lib\mysql connector-java-3.0.9-stable-bin.jar' is not found." I do not know how to read these error messages. Thank you for your help.*
P2 at 18:15, 21 December 2007: *That will be a true cause!! Please append a path of the jar file of the mysql connector to the build path of your project. You can paste the jar file in the WEB-INF folder and then append the build path by an operation of clicking right button on the project.*

Questioner (S1) at 19:13, 21 December 2007: *Two mysql connectors existed in my project. When I deleted one of them, the problem has resolved!! As I could not resolve this problem by myself, thank you very much for your support.*

**(2) TA and instructor as knowledge broker**

In spite of problem resolution information being registered in the system, some students failed to find it out that helped to resolve their problem. The teaching staff played a role of knowledge broker to inform them of the applicable information. The following is examples:

Example 1

Student S2: *I do not know how to implement a function of sending a mail, although I investigated the way.*

Instructor: *The problem resolution information registered at 20:26, 22 January 2008 may help you. Please refer to the information. I clearly remember that the student who registered this information spent many efforts and finally resolved the problem.*

Example 2

Student S3: *After compiling a system, I designated a URL in my web browser. However the 404 error happened.*

A teaching assistant: *You can find solution by accessing the problem resolution information registered at 09:34, 1st November 2007.*

**(3) State of reuse of the problem resolution information**

From the feedback for the problem resolution information, we found at least seven problem resolution information was reused to solve problems of some students (how to backup and restore a database, control from a sub-page to a main page, automatic mail distribution, and association class). However, it is difficult to ascertain true state of reuse of the information, because users have to visit the page and evaluate it after their problem has fixed (they may not come back the page after fixing their problem).

## 4.3 Discussions

### 4.3.1 On problem resolution information

Similar trend on the number of registered problem resolution information, ratio of classification by phase and by contents was seen in both years. Problem resolution information on coding with respect to phase was registered most. In this course, students are assigned to different use cases. Thus technologies required differ respectively. A problem of a student must basically be resolved by him/herself. Therefore I think problems seem to be overt. On the other hand, design is group work in this course. This course also exposes design inspection. Groups are required to revise artifacts according to the inspection comments. It may be difficult to abstract from inspection comments to know-how. Much of the problem resolution information in the design phase was that on how to use tools (UML editor) (in 2007, two out of three problem resolution information and in 2008, two out of six were that on how to use tools).

According to analysis of the registered problem resolution information from the viewpoint of contents, most information was that on technical. In 2007, two students who have less confidence on technical aspects registered the problem resolution information as follows: "*It is important to possess an attitude that asks how to investigate something, not that asks for the direct solution for the problem you face*", "*You should not give up even though you can't understand. Participate in your project in a positive manner, and your team members support you when you have troubles.*" They were on the process aspect how a problem should be resolved.

### 4.3.2 On knowledge broker

In this case study, we observed knowledge transfer by the teaching staff in the discussion space and in the BBS of a group.

Boden and Avram studied knowledge distribution between sites of geographically distributed software projects in small companies [3]. They concluded importance on oral communications by means of usage of Skype, business trips, bridging knowledge by developers who stay in another site.

On the other hand, our course is not geographically distributed, rather temporally distributed because students have their different schedules. Therefore synchronous communications are limited. People (developers and the teaching assistants) also change off very fast in the course. Oral communications are not suitable for this context because it is volatile. For these reasons, we adopted a method to describe, store, and share problem resolution information. Nakakoji et al. [8] proposed nine items as a design guideline for expertise communication support. They described that if documents or codes exist to obtain some information, they should be used as much as possible so that communications do not occur. Our system is satisfied with this item of the guideline.

## 5. Conclusion

This paper has analyzed state of the problem resolution information sharing in a software engineering project course from two years' usage of the problem resolution information sharing system. Similar trend on the number of registered problem resolution information, ratio of classification by phase and by contents was seen in both years (problem resolution information on coding with respect to phase was registered most, and most information was that on technical with respect to contents). In this case study, we observed knowledge transfer by the teaching staff in the discussion space and in the BBS of a group.

We would like to ascertain the status of reuse of the problem resolution information as future work.

## Acknowledgement

The authors would like to thank anonymous reviewers for their comments to improve this paper. They also would like to thank Mr. Hiroaki Yamada for his help to gather log data of the system usage.

## References

1. M. S. Ackerman, and T. W. Malone, Answer Garden: a tool for growing organizational memory, Proceedings of the ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems, pp.31-39, ACM Press, 1990.
2. A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, Supporting Online Problem-Solving Communities with the Semantic Web, Proceedings of the 15th World Wide Web Conference (WWW2006), pp.575-584, ACM Press, 2006.
3. A. Boden, and G. Avram, Bridging knowledge distribution - The role of knowledge brokers in distributed software development teams, Proceedings of the ICSE 2008 Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2009), ACM Press, 2009.
4. Y. Hatamura, Learning from Failures, Kodansha, 2000 (In Japanese).
5. A. Hazeyama, K. Shimada, and Y. Kobayashi, A Collaborative Problem Solving Support System for Group-based Software Engineering Project Course and Its Application, Proceedings of the Seventh International Conference on Creating, Connecting and Collaborating through Computing (C5 2009), pp.74-78, IEEE Computer Society Press, Kyoto, Japan, January 2009.
6. A. Ishida, et al., Research on the Problem in Information-sharing and Its Efficiency in Software Development, (In Japanese).
7. M. Miura, Y. Kobayashi, K. Shimada, K. Takahashi, S. Seiki, and A. Hazeyama, A Proposal of Integrating Personal and Community Support with Learning Environment for Group-based Software Engineering Course, Proceedings of the 2nd International Conference on Knowledge, Information and Creativity Support Systems (KICSS 2007), pp.144-151, 2007.
8. K. Nakakoji, Y. Ye, and Y. Yamamoto, Interaction Design for Supporting Knowledge Communication in Software Development, Annual Conference of the Japanese Society for Artificial Intelligence (JSAI 2009), Takamatsu, Japan, June, 2009 (In Japanese).
9. N. Serrano, and I. Ciordia, Bugzilla, ITracker, and other bug trackers, IEEE Software, Vol. 22, No.2, pp.11-13, 2005.
10. Y. Ye, Socio-Technical Support for Knowledge Collaboration in Software Development Tools, Proceedings of the Workshop on Integrating Software Engineering and Usability Engineering, pp.39-51, 2005.

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

# On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study

Emad Shihab Nicolas Bettenburg Bram Adams Ahmed E. Hassan

Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, K7L 3N6, Canada
{*nicbet, emads, bram, ahmed*}*@cs.queensu.ca*

**Abstract.** Mailing lists provide a rich set of data that can be used to improve and enhance our understanding of software processes and practices. This information allows us to study development characteristics like team structure, activity, and social interaction. In this paper, we perform an exploratory study on the GNOME project and recover operational knowledge from mailing list discussions. Our findings indicate that mailing list activity is driven by a dominant group of participants, that it is greatly connected to development activity, yet influenced by external factors like market competition. Our results provide a broad picture of the central role played by mailing lists in open source projects.

## 1 Introduction

Most open source developers communicate through mailing lists. This style of communication makes mailing lists a rich source of information which researchers can use to understand software processes and improve development practices. Mailing lists have been used to infer social structure [4, 5, 11], identify architectural changes [1], and most recently to study the code review process [3, 14, 18].

However, understanding the generality of the results derived from mailing lists requires that we first understand how mailing lists are used in practice and the impact of their usage patterns on the information in the lists. For example, previous studies (e.g. [4, 5]) studied the social structure of developers using mailing lists, however, does this social structure change over time? How fast does the structure change?.

The central role played by mailing lists is depicted in Figure 1. Developers use mailing lists to discuss a variety of issues and project decisions [1, 10]. Many of these issues and decisions are related to and affect the source code. These issues are often driven by external factors such as the introduction of new features in competing products.

In this paper, we perform an exploratory study on the role played by mailing lists. Performing an exploratory study on mailing lists provides a holistic view of their role. This holistic view enhances the understanding of the findings of in-depth studies, unveils details which may not be apparent through in-depth studies and helps identify interesting directions for future research.

To perform our study, we use the mailing lists from 22 GNOME projects. The study centers around the following aspects, shown in Figure 1, in an open source project:

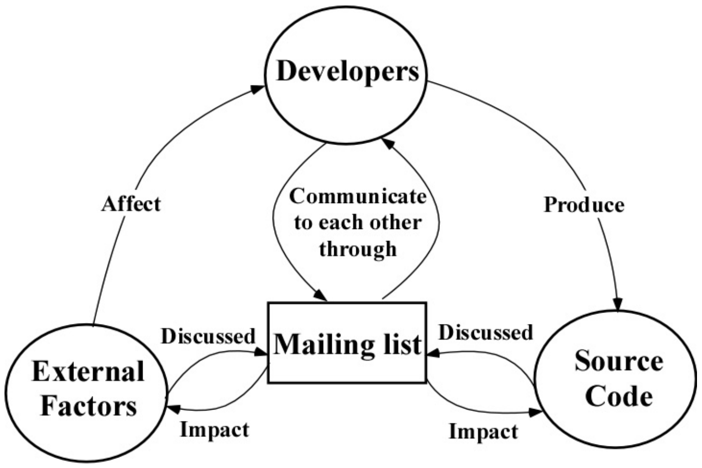


**Fig. 1.** The central role of Mailing lists in open source projects

| | | Number of | | | | |
|---|---|---|---|---|---|---|
| **Project name** | **Start date** | **Messages** | **Participants** | **Age (months)** | **Threads** | **Application Domain** |
| Deskbar Applet | Oct-05 | 1,098 | 106 | 39 | 340 | Search interface |
| Ekiga | Aug-06 | 5,389 | 690 | 29 | 1,200 | Teleconferencing |
| Eog | Mar-01 | 458 | 106 | 93 | 233 | Image viewer |
| Epiphany | Dec-02 | 5,735 | 905 | 73 | 1,608 | Web browser |
| Evince | Jan-05 | 1,358 | 415 | 48 | 566 | Document viewer |
| Evolution | Jan-00 | 53,927 | 6,026 | 96 | 15,718 | Email client |
| Games | Feb-03 | 1,590 | 190 | 71 | 531 | Computer games |
| Gdm | Mar-00 | 2,578 | 675 | 105 | 1,040 | Display manager |
| Gedit | Apr-00 | 2,237 | 530 | 104 | 919 | Text editor |
| Multimedia | Oct-00 | 1,646 | 273 | 98 | 507 | Multimedia library |
| Network | Aug-03 | 673 | 105 | 65 | 267 | Network tools |
| Power Manager | Jan-06 | 1,059 | 199 | 36 | 305 | Power management |
| Themes | Jan-98 | 1,310 | 221 | 132 | 447 | Window manager |
| Utils | Oct-04 | 358 | 106 | 51 | 279 | Utility applications |
| Control Center | Dec-99 | 1,478 | 168 | 97 | 311 | Configuration |
| Libsoup | May-06 | 83 | 24 | 32 | 41 | HTTP library |
| Metacity | Sep-05 | 262 | 48 | 40 | 59 | Window manager |
| Nautilus | Apr-00 | 22,488 | 2,384 | 105 | 5,582 | File manager |
| Orca | Jan-06 | 11,930 | 516 | 36 | 3,598 | Screen reader |
| Screensaver | Oct-05 | 139 | 25 | 39 | 30 | Screensaver |
| Seahorse | Jun-07 | 252 | 34 | 19 | 116 | Encryption management |
| System tools | Nov-99 | 1,832 | 327 | 98 | 792 | System admin tools |

**Table 1.** General overview of the GNOME mailing lists studied

– **Developers:** We characterized the communication style of mailing list's participants, i.e., the developers from the development mailing lists.

  *We found that a small number of developers play a central role in driving the mailing list activity. We also found that these developers remain stable throughout the lifetime of a project.*

– **Source code:** We explored the impact of mailing list activity on the source code activity, i.e., changes.

  *We found that there is a high correlation between mailing list activity and source code activity.*

– **External Factors:** We examined the effect of external factors, such as competing products on mailing list activity.

  *We found that competing products shape and drive many of the discussions on mailing lists.*

**Overview of Paper.** The rest of the paper is organized as follows: Section 2 discusses the motivation for using the GNOME project as a case study and presents statistics about the project. We present and analyze our findings in Section 3. The threats to validity are discussed in Section 4 and the related work is presented in Section 5. Section 6 concludes the paper.

## 2 GNOME as a Case Study

In this section, we detail the case study project used in our study. The GNOME project is composed of approximately two million lines of code and has more than 500 different contributors from all over the world [9]. The GNOME project is composed of many small projects that cover a wide range of applications, e.g., email client, text editor, and file manager. The main source of communication for GNOME developers is the developer mailing list for each project. These projects vary in size, age, user and developer base. We expect these differences in size, age, and domain to have an impact on the mailing lists of these projects. Therefore, studying the mailing lists of the different projects can lead to interesting and generalizable findings and open new directions for future research.

Table 1 presents a general overview of the mailing lists used for this study. The `Project name` column lists the name of the GNOME module. The `Start date`, `Number of Messages`, `Number of Participants`, `Age` and `Number of Threads` columns list the month and year of the first commit to the project's trunk (derived by examining the source control repository for the project), the total number of messages, the number of participants, the age, and the number of threads of the GNOME projects, respectively. In addition, the `Application Domain` column lists the application of the project. All calculations are based on the participation from the start date listed till the end of 2008, inclusive.

## 3 Results and Analysis

We now study the three aspects outlined in Figure 1 using the GNOME mailing list data. Subsections 3.1 and 3.2 cover the developers aspect, subsection 3.3 covers the source code aspect and subsection 3.4 covers the external factors aspect. We start each subsection by presenting our motivation to explore the aspect. We then describe the approach that we used to perform our exploration. Finally, we present our results and outline our main findings.

Since most of the GNOME mailing lists have low activity, we will often use the Evolution and Nautilus projects to more closely explore many of our findings since the two projects account for more than 65% of the total messages. We highlight the results that generalize for the rest of the 20 projects, where applicable.

### 3.1 Communication Style in Mailing Lists

*Is mailing list activity mostly driven by a few participants (a dominant group) or is the participation evenly distributed? Does the dominant group engage in discussions with others or is it mostly involved in internal discussions?*

**Motivation.** The Pareto principle (also known as the 80-20 rule), which states that the majority of the effects come from a minority of the causes, has applications in many fields. For instance, research shows that 20% of the code contains 80% of the bugs [8]. We hypothesize that there exist a few key participants (who we call the *dominant group*) in mailing lists, that are responsible for most of the messages posted on the mailing list. Most likely, they are members who are very knowledgeable about the project and use their knowledge to support newcomers and casual participants (who we call the *casual group*). It is important for us to investigate whether these experts exist on mailing lists for two reasons: 1) one can address his/her questions directly to such experts to receive a more accurate and speedy response and 2) the discussions of these experts can be used for future reference by others who are less knowledgeable about the project.

In addition, if such a group exists, we would like to know if they actively engage in discussions with others who are outside of the dominant group. If in fact they do engage with others then we can safely assume that newcomers and less experienced developers will benefit from these experts. If we determine otherwise, i.e. that the dominant group is a closed group, then newcomers and other participants may be better off reading previous discussions and learning from them rather than attempting to establish direct contact with the dominant group members.

**Approach.** We measured the number of messages contributed by the top 10% most active participants, who we call the dominant group. We found evidence that in fact there does exist a dominant group for each of the 22 GNOME mailing lists. The dominant group contributes a large amount of the messages posted.

Then, we examined the active discussion threads and classified these active threads into threads with:

- **Dominant group members only:** A high number of such threads implies that the dominant group is a closed group that does not engage with others.

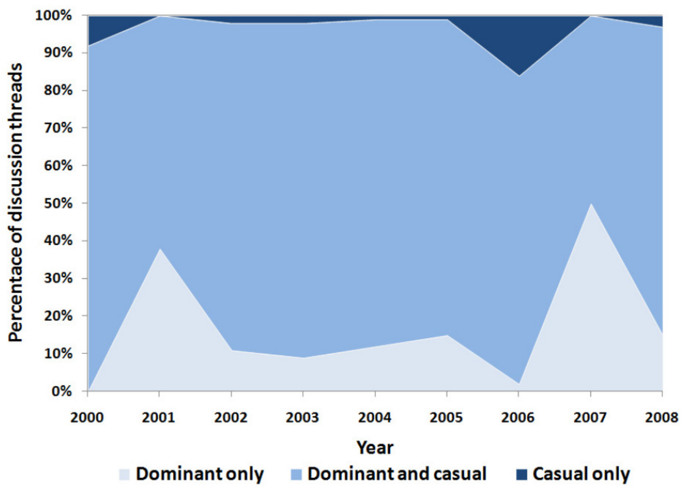


**Fig. 2.** Distribution of discussion types in the Evolution project

- **Dominant and casual members together:** A high number of such threads is a good indicator of a stimulating mailing list where expert and casual participants actively engaging in discussions.
- **Casual group members only:** A high number of this type of discussion would indicate that the casual members are not integrated into the mailing list.

**Results.** In addition to finding out that there exists a dominant group in each mailing list, we quantified their contribution. We found that on average the dominant group accounts for approximately $60\%$ of the messages. *This finding is consistent across all of the 22 GNOME projects*. We did not observe a consistent finding when we considered the top $20\%$ of the participants (i.e. we did not find evidence of the Pareto principle).

We plot the number of threads for the Evolution and Nautilus projects in Figures 2 and 3, respectively. In both projects, we found that the majority of the active discussions involve dominant and casual group members. On average, in $82\%$ of the discussions dominant and casual group members were present. In $16\%$ of the discussions, dominant group members were discussing exclusively and in the remaining $2\%$ of the discussions the casual members discussing exclusively. We believe that it is a sign of a productive mailing list when the two groups actively engage in discussions, with the dominant group members most likely playing a supporting role for the casual group members.

However, in some cases a high percentage in discussions that involve dominant and casual members may not be desired. For example, some dominant group members may be overwhelmed by a high number of questions from casual members (since casual members may make unreasonable requests from more knowledgeable dominant group members). Whether a high number of discussions between casual and dominant group members is indicative of a productive mailing list depends on the product domain and the mailing list's members' knowledge.

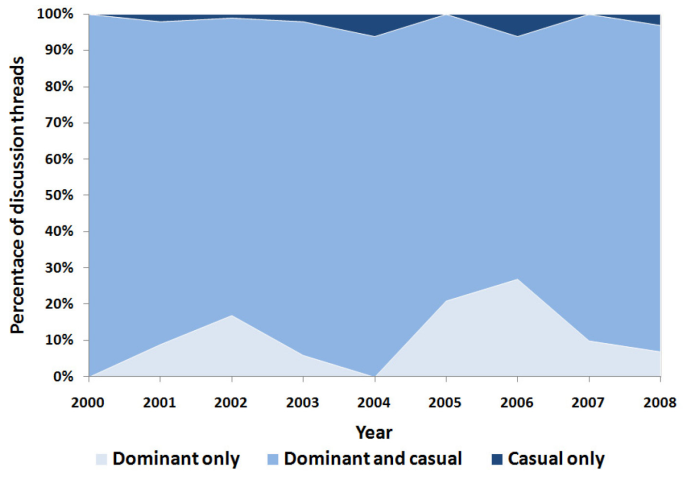


**Fig. 3.** Distribution of discussion types in the Nautilus project

*10% of mailing list participants (the dominant group) contribute 60% of the messages in a mailing list. The dominant group is very active and is engaging with outside-members, i.e. casual members.*

## 3.2 Stability of Mailing List Participants

*Do dominant group members change over time? If so, how much are they changing by? How is their stability compared to rest of the mailing list participants?*

**Motivation.** As we have seen in the previous subsection, the dominant group plays an important role in the mailing list. They contribute the majority of messages posted and are involved in approximately 96% of active discussions. For this reason, it is quite important that dominant group members do not change frequently. We study the stability of the dominant group. In particular, we measure the variation in the dominant group over time. A relatively stable dominant group (i.e. one that does not change frequently) is desirable because it means that dominant group members spend enough time in the project and achieve a higher level of expertise to better support casual group members.

**Approach.** To measure the stability of members in the dominant group, we performed two studies:

- **Dominant group change over time:** We measured the change between two consecutive years. This gives us a measure of how much a dominant group changes by from one year to the next.
- **Dominant group change compared to casual group change:** We measured the change of the casual group for two consecutive years and compared it to the change in the dominant group.

| | Evolution | | Nautilus | |
|---|---|---|---|---|
| **Year** | **Dominant** | **Casual** | **Dominant** | **Casual** |
| 2000 - 01 | 0.68 | 0.11 | 0.73 | 0.20 |
| 2001 - 02 | 0.74 | 0.11 | 0.55 | 0.20 |
| 2002 - 03 | 0.63 | 0.16 | 0.40 | 0.21 |
| 2003 - 04 | 0.74 | 0.16 | 0.85 | 0.23 |
| 2004 - 05 | 0.84 | 0.16 | 0.76 | 0.24 |
| 2005 - 06 | 0.70 | 0.19 | 0.95 | 0.24 |
| 2006 - 07 | 0.35 | 0.17 | 0.88 | 0.19 |
| 2007 - 08 | 0.80 | 0.15 | 0.77 | 0.16 |
| **Average** | **0.69** | **0.15** | **0.73** | **0.21** |

**Table 2.** Cosine distance of dominant and casual groups of the Evolution and Nautilus projects

We used the Cosine Distance (CD) similarity metric to measure the similarity between the groups in two consecutive years. The CD metric outperforms other simple measures such as intersection or proportion which only measure the existence of a participant but not their level of contribution. The CD similarity is defined as:

$$CD(P,Q) = \frac{\sum_x P(X)Q(X)}{\sqrt{\sum_x P(X)^2}\sqrt{\sum_x Q(X)^2}}, \tag{1}$$

where $P(X)$ and $Q(X)$ represent the two input distributions to be compared. A value of 0 for the CD metric means that the group has changed drastically across two years with no members in common. A value of 1 for the CD metric indicates that the group is the exact same (i.e. is it a very stable group).

The Cosine Distance metric takes as input two participation distributions – one for each of the years under study. Each distribution has the contribution of each of the participants for that year. So when comparing the dominant group for the year 2000 and year 2001, the 2000 and 2001 participation distribution for the dominant group is used. One major challenge we faced when conducting this study was the use of multiple aliases by developers [4]. We used heuristics based on regular expressions to address this challenge as detailed in our previous work [2].

**Results.** The calculated CD values for the Evolution and Nautilus projects are shown in Table 2. It is observed that the dominant group is more stable than the casual group. On average, the dominant group is 3 times more stable than the casual group. *These two findings are observed across all of the 22 GNOME projects*. The same stability of social structures were also observed with the FLOSS projects [22]. This is a positive sign about the health of the dominant groups of many of these projects. Dominant group members, who are critically important to the mailing list of the project are stable enough to pass their knowledge to newcomers and casual group members.

| | Type of change | | |
|---|---|---|---|
| **Project** | **Add** | **Remove** | **Modify** |
| Evolution | **0.83** | 0.60 | 0.61 |
| Nautilus | 0.32 | 0.53 | **0.85** |

**Table 3.** Correlation between the number of messages per year and the type of source code change

*The participants in the dominant group are very stable over time. On average, they are about 3 times as stable as casual participants.*

## 3.3 Source Code Activity and Mailing List Activity

*Can mailing list activity be used to infer information about source code activity (amount of work done on the source code)?*

**Motivation.** Since mailing lists are the main source for developer communication [10], we expect that mailing lists contain useful information about the source code of a project. We want to explore if we can infer the types of source code changes and the level of activity done on the source code through the mailing list activity. Because developers often use the mailing list to discuss their source code changes and get assistance or feedback on these changes [14], we hypothesize that there will be high correlation between the mailing list activity and the code activity. Or in other words, the more work done on the source code, the more it will be discussed on the mailing list and vice-versa.

**Approach.** We mined the SVN source control repository and extracted the number of lines added, removed and modified per year for each project. We defined a Code Activity (CA) metric, defined as:

$$CA(Y) = A_Y + R_Y + M_Y, \quad (2)$$

where $A_Y$, $R_Y$ and $M_Y$ refers to the number of lines of source code added, removed and modified in year $Y$, respectively. We used this metric and measured the correlation between it and the mailing list activity, i.e., the number of messages per year. Furthermore, we examine the correlation between the number of messages and the type of the performed change (add, delete, modify).

**Results.** The number of messages per year and the Code Activity for the Evolution and Nautilus projects are plotted in Figures 4 and 5, respectively. It can be observed that there is a high correlation between the number of messages on the mailing list and the Code Activity metric. This finding shows that developers do rely heavily on the mailing list to discuss source code changes. As for the correlation between the level of mailing list activity and the type of change, we present the results in Table 3. We found that in the Evolution project, the highest correlation was between the number of

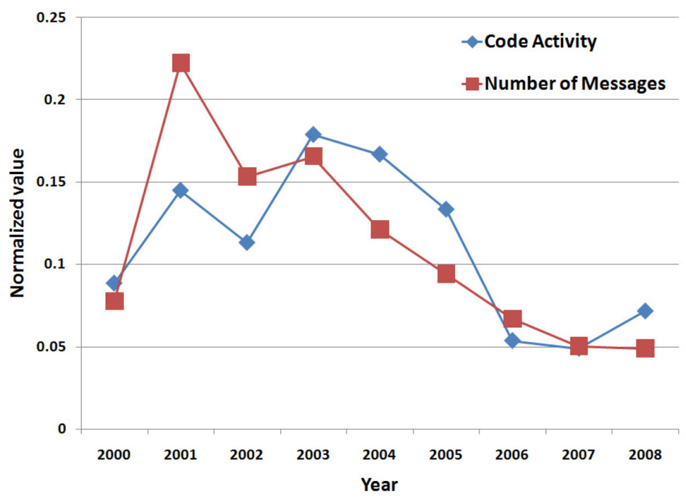


**Fig. 4.** Number of messages and Code Activity for the Evolution project

messages and the lines of code added ($\rho = 0.83$). On the other hand, in the case of the Nautilus project, we found that the highest correlation is between the number of messages and the lines of code modified ($\rho = 0.85$). It seems that in the Evolution project, participants are discussing code additions more than they are discussing code removal or modifications, while for the Nautilus project, code modifications are being discussed more than code additions and removals. We believe that further investigation is needed here to better understand the rationale for this discrepancy between both projects and whether it indicates different development and communication styles.

To verify, we measured the occurrence of terms that indicate code additions and code modifications in the mailing lists of the two projects. Since most commonly, code additions involve the introduction of new features, we classified the terms "new features" and "feature request" as indicators of code additions. Code modifications are usually carried out to fix bugs which are found during the testing phase and applied via patches. For this reason, we associate the terms "bug", "patch", "testing", and "maintain" to code modifications. We observed that in the Evolution mailing list, the terms associated with the addition of new features were mentioned in $57\%$ more messages than on the Nautilus mailing list. On the other hand, the terms associated with code modifications were mentioned in $75\%$ more messages in the Nautilus mailing list compared to the Evolution mailing list. The findings are consistent with our correlation results shown in Table 3.

*Mailing list activity is closely related to source code activity. In addition, mailing list discussions are good indicators of the types of source code changes being carried out on the project*

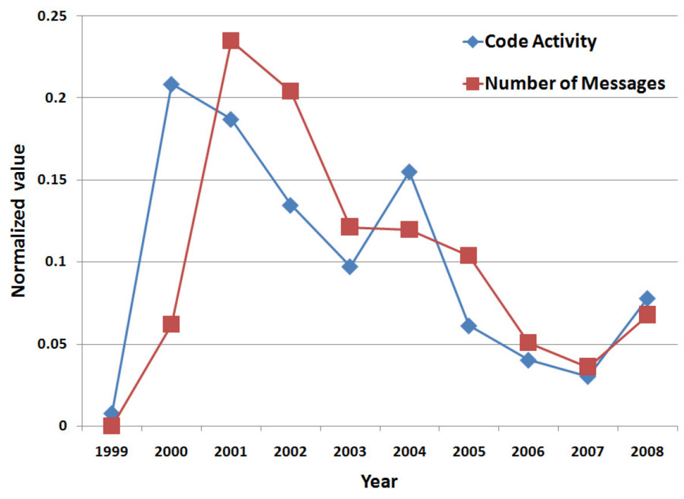


**Fig. 5.** Number of messages and Code Activity for the Nautilus project

## 3.4 Effect of External Factors on Mailing List Activity

*Can we observe the effect of external factors on mailing list activity?*

**Motivation.** One of the benefits of studying mailing lists is that they can provide us with knowledge about issues that indirectly affect a project, i.e., external factors. Market competition and management changes are examples of external factors. Such knowledge about external factors is often hard to uncover as it is not recorded in the source code or documentation. However, this knowledge is very important since it helps explain certain observed behaviors, such as an increase in bugs or the lack of interest in a project (and maybe its eventual death). We attempt to observe the effect of external factors on mailing list activity.

**Approach.** Due to space limitation, we perform the study of external factors on the Evolution project only. However, we note that our approach can be applied to any other project. We study the mailing list activity trend and perform two types of analysis: quantitative and qualitative analysis. In the quantitative analysis study, we treat the bodies of all email messages as a bag-of-words and compare the occurrence of the names of competing mail clients ("gmail", "outlook", and "thunderbird") to the occurrence of the terms: "evolution" and "evo" (a short hand form often used to refer to the evolution project). A rise in the number of times a term occurs indicates that it is being discussed more, hence it has a greater impact. In the qualitative study, we read through several email postings to better understand and clarify our quantitative findings.

**Results.**

*Quantitative analysis:* Looking at Figure 6, we observe that the activity on the Evolution mailing list is increasing from 2000 to 2001. This increase can be attributed to the creation of Ximian at the end of 1999, which was created to continue the development

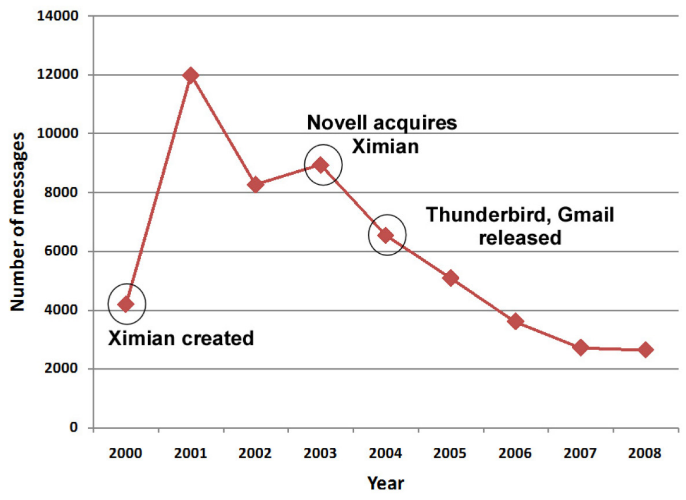


**Fig. 6.** Messages per year on the Evolution mailing list

of the Evolution project [9]. This acquisition increased the attention and support for the Evolution project, hence the continuing increase in mailing list activity.

Then, from the year 2001 on, we observe a steady decline in mailing list activity (except for a small increase in activity in the year 2003). Market competition, along with organizational changes may have caused this decline. The results of the quantitative study (which measures the frequency of occurrence of terms in the message bodies per year) are shown in Figure 7. We observe a steady decrease in the use of the terms “evolution” and “evo”, suggesting that the Evolution project is being discussed less frequently. At the same time, there is a steady increase in the number of times its market competitors “gmail”, “outlook” and “thunderbird” are being mentioned.

*Qualitative analysis:* We read through several mailing list posting to better understand our aforementioned quantitative findings. The following quotations are excerpts from discussions that took place when a declining level of activity was observed:

> “*...Furthermore, I can’t find where in the Tools menu to change this: the option is no longer present on any of the dialog boxes. Which is why I’m sending this with Thunderbird...*”
> “*...Unless Ximian implements some features that aren’t important to Ximian but are important to its users, evo will be relegated to ”toy” status. I’m currently struggling to remain with my current distro of SuSE+Ximian in my business, but the lack of meaningful support in both components is forcing my hand to look around for another solution...*”

We believe that these excerpts show that the Evolution mail client was and is losing market share due to competition from other competing mail clients, such as Thunderbird, with many of the postings pointing people to competing products.

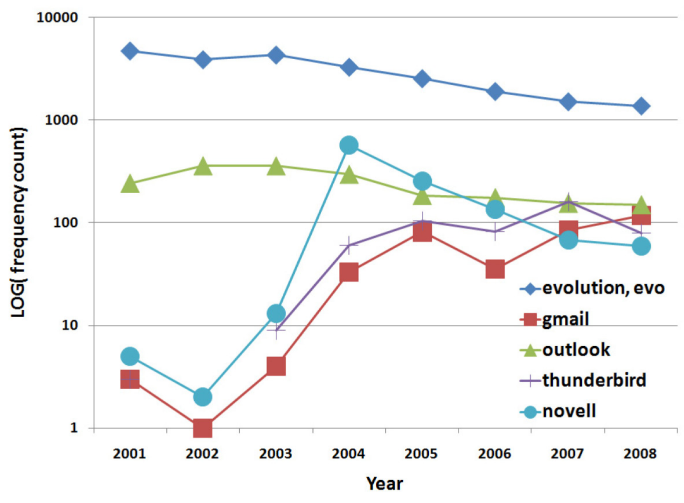


**Fig. 7.** Frequency of terms in the Evolution mailing list

As for the spike in activity on the Evolution mailing list in the year 2003, we believe this can be attributed to Novell's acquisition of Ximian in late 2003 [9]. We counted the occurrence of the term "novell" in the mailing list and found that the number of times the term "novell" was mentioned on the Evolution mailing list spiked from 13 in 2003 to 574 in 2004 (as depicted in Figure 7). This spike is most likely due to hype surrounding Novell's acquisition, which quickly dies off in the coming years.

This study on external factors suggests that mailing lists can be leveraged to study the effect of external factors on a project. Furthermore, such information can be used to explain design decisions that happened in the past.

*External factors affect mailing list activity.*

## 4 Threats to Validity

In our stability analysis, we used the names of developers as identifiers. Although we used heuristics to resolve multiple aliases [2] (i.e. participants who use multiple email address and names), we were not able to deal with some rare cases. Additionally, in our study we assume that all mailing list participants are developers. This assumption is true for the vast majority of the cases (especially since we are considering developer mailing lists), but in some cases, it is possible that a participant on the developer mailing list is not engaged in any developmental effort.

In our studies on source code activity and external factors, we measure the frequency of key terms that we associate with specific topics (i.e. the term "maintain" with the topic maintenance). Although our list is not exhaustive and does not contain all the terms that may be associated with the respective topic, we believe that the terms used in our study are the most common and cover the majority of the terms that would be used to refer to the topic.

Finally, our findings may not generalize to all open source projects.

## 5 Related Work

Previous work used mailing lists to study the social structure of developers. Bird *et al.* [4, 5] used mailing lists to study the social networks created by developers and non-developers. In their follow-on work [7], they extracted the sub-community structure from these social network and studied their evolution over time. Ogawa *et al.* [12] used Sankey diagrams to visualize evolving networks in mailing lists and concluded that social behavior can be related to events in a project's development.

In addition, several studies used mailing lists to study developer morale, work times and the code review process. Rigby and Hassan [15] performed a psychometric study on the Apache httpd mailing list to identify the personality types of open-source software developers and gain insight on the level of optimism in pre- and post release phases. Tsunoda *et al.* [17] used mailing lists to analyze developer work times and found that the ratio of committer messages sent during overtime periods is increasing every year. Weissgerber, Neu and Diehl [18] used mailing lists to study the likelihood of a patch getting accepted.

Furthermore, other studies used mailing lists to study developer coordination, motivation and knowledge sharing. Yamauchi *et al.* [19] studied the coordination mechanisms used by OSS developers to achieve smooth coordination. They found that spontaneous work coordinated afterward is effective, rational organizational culture helps achieve agreement among OSS members and communications media, such as CVS and mailing lists, moderately support spontaneous work. Lakhani and von Hippel [21] used mailing lists to study the motivating factors of OSS participants to perform mundane tasks. They found that direct learning benefits is one of the main motivators for these participants to conduct such tasks. Sowe *et al.* [20] studied knowledge sharing between developers in mailing lists. They found that developers share knowledge a lot.

Other work combined the information extracted from mailing lists with information from other repositories (e.g. the source code repository). Robles and Gonzalez-Barahona [16] used information from multiple historical archives to assist in accurately identifying actors. Baysal and Malton [1] used the similarity between mailing list and source code archives to identify architectural changes. Bird *et al.* [6] combined the use of mailing lists and the source code repository to study the time it takes for developers to be invited into the core group of a project.

Our work recognizes the central role played by mailing lists and, to the best of our knowledge, is the first to perform an exploratory study using a large number of mailing lists. The study on the communication style of participants and their stability is novel and complements previous work. For example, previous work on social network analysis, developer morale, work times and evolution could have treated dominant and casual group differently and put more emphasis on the dominant group findings. Doing so would enhance the impact of their findings and provide a better understanding of the phenomena being observed. The findings from our source code activity and external factors studies can assist researchers who use mailing lists in combination with source code repositories (e.g. [1, 13]) better understand the relationship between the two. Further, taking into account the effect of external factors may help explain some unexpected observations.

## 6 Conclusions

In this paper, the central role of mailing lists was studied through an exploratory study. The study centered around three aspects: developers, source code and external factors.

Our findings indicate that a small number of participants (dominant group) account for the majority of the messages posted on mailing lists. The dominant group is very active and engaging with others and its composition is very stable (3 times more stable than casual members). In addition, we found that mailing list activity is closely related to source code activity and mailing list discussions are good indicators of the types of source code changes being carried out on the project. Lastly, we showed that external factors affect mailing list activity.

## References

1. O. Baysal and A. J. Malton. Correlating social interactions to release history during software evolution. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 7, 2007.
2. N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using of off-the-shelf techniques to process mailing list data. In *ICSM'09: Proceedings of the 25th International Conference on Software Maintainance*, 2009.
3. C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.
4. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, 2006.
5. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks in postgres. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 185–186. ACM, 2006.
6. C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? immigration in open source projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6, 2007.
7. C. Bird, D. Pattison, R. D'Souza, V. Folkiv, and P. Devanbu. Latent Social Structure in Open Source Projects. In *FSE '08: Proceedings of the 2008 ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 24–35, 2008.
8. B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.
9. D. M. German. The gnome project: a case study of open source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, September 2004.
10. D. M. German. Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maint. Evol.*, 16(6):367–384, 2004.

11. L. Hossain, A. Wu, and K. K. S. Chung. Actor centrality correlates to project based coordination. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 363–372, 2006.
12. M. Ogawa, K.-L. Ma, C. Bird, P. Devanbu, and A. Gourley. Visualizing social interaction in open source software projects. *Asia-Pacific Symposium on Visualization*, 0:25–32, 2007.
13. D. Pattison, C. Bird, and P. Devanbu. Talk and work: a preliminary report. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 113–116, 2008.
14. P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 541–550, 2008.
15. P. C. Rigby and A. E. Hassan. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 23, 2007.
16. G. Robles and J. M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.
17. M. Tsunoda, A. Monden, T. Kakimoto, Y. Kamei, and K.-i. Matsumoto. Analyzing oss developers' working time using mailing lists archives. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 181–182, 2006.
18. P. Weissgerber, D. Neu, and S. Diehl. Small patches get in! In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76, 2008.
19. Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: how open-source software succeeds. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 329–338, 2000.
20. S. K. Sowe, I. Stamelos, and L. Angelis. Understanding knowledge sharing activities in free/open source software projects: An empirical study. *J. Syst. Softw.*, 81(3):431–446, 2008.
21. K. R. Lakhani, E. von Hippel, and K. R. Lakhani. How open source software works: Free user-to-user assistance. *Research Policy*, 32:923–943, 2003.
22. J. Howison, K. Inoue, and K. Crowston. Social dynamics of free and open source team communications. In *Second Intl Conf on Open Source Systems*, pages 319–330, June 2006.

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

# A Time-Lag Analysis toward Improving the Efficiency of Communications among OSS Developers

Masao Ohira, Kiwako Koyama, Akinori Ihara,
Shinsuke Matsumoto, Yasutaka Kamei, and Ken-ichi Matsumoto

Graduate School of Information Science,
Nara Institute of Science and Technology,
8916-5, Takayama, Ikoma, Nara, Japan
{masao,kiwako-k,akinori-i,shinsuke-m,yasuta-k,matumoto}@is.naist.jp
http://se.naist.jp/

**Abstract.** Open source software (OSS) is developed by globally distributed developers with a variety of lifestyles. In such the development environment, the time-lag of communications among developers is more likely to happen due to the time difference among locations and the difference of working hours for OSS development. A means for effective communications among OSS developers has been increasingly demanded in recent years, since even an OSS product and its users requires a prompt response to issues such as defects and security vulnerabilities. In this paper, we propose an analysis method for observing the time-lag of communications among developers in an OSS project and then facilitating the communications effectively. We have conducted a case study in which our analysis method was applied to mailing-list data of the Python project. As the results, we have confirmed that our method could identify the existence of the time-lag in communications among Python developers and have achieved findings on the optimum timing for the communications.

**Key words:** time-lag analysis, distributed software development, open source software, OSS community

## 1 Introduction

Open source software (OSS) such as Linux and Apache is generally developed by globally distributed developers. Unlike commercial software development in a company, OSS development does not necessarily request developers to engage in development at a designated time and location. OSS developers may voluntarily decide whether they continue to dedicate themselves to OSS development or not.

In this OSS development environment, a time-lag occurs in communications among developers more than a little, because of differences of time zones among geographically-distributed developers with a variety of lifestyles. For instance, according to the geographical distribution of registered users at SourceForge[1] which was reported by Robles and Gonzalez-Barahona [1], the top three regions by the number of registered developers at SourceForge are North America, West Europe, and China. Since the time-lag among those regions is at least more than five hours, it would not be easy to discuss among developers in real-time. Furthermore, even if developers reside in the same time zone, it is not still guaranteed that developers can communicate each other in real time, because each developer has no constraint on working hours.

While the importance of decision-making and consensus building through discussions among developers is increasing especially in a large-scale OSS project with a number of developers, communications among developers with various time zones and lifestyles might trigger an occurrence of a time-lag and then impede rapid OSS development. In particular, in case prompt actions are required (e.g., fixing critical bugs regarding security vulnerability), the delay of decision-making and consensus building due to the communication time-lag among developers would results in decreasing software reliability and loosing users' trust.

The goal of our research is to construct a support mechanism for effective communications among geographically-distributed OSS developers. As a first step toward achieving the goal, in this paper we present an analysis method for helping OSS developers comprehend a whole picture of a communication time-lag occurred in a OSS project. The analysis method targets mailing list archives as a data source, and consists of three kinds of analyses as follows;

1. analysis of a geographical distribution and activity time of OSS developers
2. analysis of a distribution of time required for information exchanges among OSS developers in different locations, and
3. analysis of appropriate timing for sending messages.

From a case study with Python project [2] data, this paper explores the usefulness of the analysis method.

## 2 Analysis Method

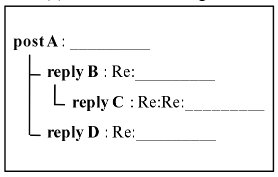
This section describes data extraction, conversion and classification which are necessary in advance of performing our analysis.

### 2.1 Preparation

**Data extraction and conversion.** The target data source for our analysis is archives of mailing lists which are used by OSS developers to exchange information. The reason we select mailing list archives as the target data for our analysis

[1] SourceForge is one of the largest OSS development community, which provides registered projects with a variety of software development support tools such as source code management tool, bug tracking system, and mailing lists. As of February 2009, more than 230,000 OSS projects and more than two million users have been registered to SourceForge.

(a) a thread in a mailing list



(b) a list of information of posted/replied messages

| ID | posted date and time | location (time zone) | Coordinated Universal Time (UTC) |
|---|---|---|---|
| message A | 2009/04/29 10:33:53 | UTC+9 | 2009/04/29 01:33:53 |
| message B | 2009/04/28 22:40:04 | UTC−4 | 2009/04/29 02:40:04 |
| message C | 2009/04/29 09:12:30 | UTC+3 | 2009/04/29 06:12:30 |
| message D | 2009/04/29 02:26:59 | UTC−10 | 2009/04/29 12:26:59 |

(c) a relationship among posted and replied messages

| info. of posted messages | | | info. of replied messages | | | time lag (hours) |
|---|---|---|---|---|---|---|
| ID | posted date and time | location | ID | replied date | location | |
| message A | 2009/04/29 10:33:53 | UTC+9 | message B | 2009/04/28 22:40:04 | UTC−4 | 1.10 |
| message B | 2009/04/28 22:40:04 | UTC−4 | message C | 2009/04/29 09:12:30 | UTC+3 | 3.54 |
| message A | 2009/04/29 10:33:53 | UTC+9 | message D | 2009/04/29 02:26:59 | UTC−10 | 10.88 |

**Fig. 1.** Data extraction and conversion

is because mailing lists are widely used in OSS projects. We consider that data of mailing list archives allows us to reveal a whole picture of the existence of time-lag in many OSS projects.

In order to apply the analysis method to the target data, firstly we need to extract information of **posted date and time**, and **posted locations** from mailing list archives (i.e. from e-mail headers). In what follows, "posted date and time" means local date and time of a message's sender, and "posted locations" is presented as time-lag between Coordinated Universal Time (UTC) and local time. For instance, "**UTC+9**" means the location of Japan because the standard time of Japan is nine hours prior to UTC.

Figure 1 shows the procedure of data extraction and conversion. When a developer posts a message to a mailing list, the message is delivered to subscribed developers of the mailing list. Replying to the post, the other developers can discuss the message Using such the post-reply relationship (i.e., thread structure) in a mailing list, we extract information on posted/replied date and time, and locations (time zones) from mailing list archives [2].

For instance, from a thread structure illustrated in Fig.1(a), we extract information of posted and replied messages as the table in Fig.1(b). Then we convert the information into post-reply relationships as the table in Fig.1(c) and calculate time-lag from a difference between posted and replied date and time. Note that we suppose that message B replied to message A can be a posted message for message C.

**Classification of data.** Several factors such as differences of time zones (i.e., countries and/or regions) and differences of developers' working hours may have

[2] We do not collect data from posted messages with no replies.

an influence on time-lag between posted time and replied time. For instance, communications among developers living in different time zones might be prolonged because of differences of lifestyles (e.g., dinner time or sleeping time). And developers in the same time zone might be still difficult to communicate each other in real time, because each developer has no constraint on working hours.

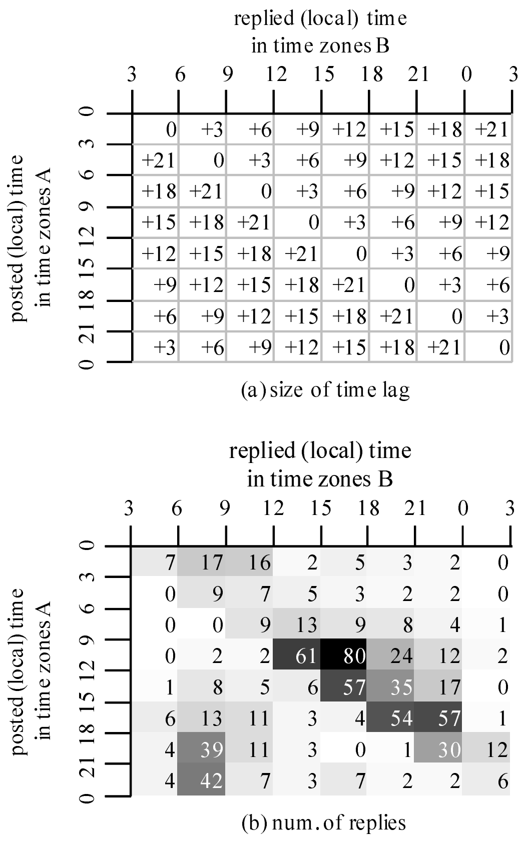
In order to distinguish between the time-lag due to time zone differences and the time-lag due to lifestyle differences, the collected data described above is classified into data **within** and **over** the time-lag of 24 hours. Many of replied messages within 24 hours after a post would be affected by differences of time zones, while replied messages over 24 hours after a post would be generated by differences of developers' lifestyles and/or difficulty of the content of a posted message, rather than geographical differences among developers. For these reasons, our analysis method targets the data of posted and replied messages within the 24 hours time-lag.

## 2.2 Procedure

**Geographical distribution and activity time of OSS developers.** In order to understand the existence of the communication time-lag in an OSS project, the analysis method firstly identifies a geographical distribution of developers of the project, counting the number of replied messages by each location (UTC−11 ∼UTC+12). The analysis method also identifies a distribution of the number of replied messages by local time in each location in oder to understand working hours of developers by each location, since developers' working hour can differ even in the same location. By this means, we can identify active or inactive locations and working hours of OSS developers.

**Distribution of time required for information exchanges among OSS developers in different locations.** In order to understand the communication time-lag due to the geographical (time zone) differences, the analysis method calculates distributions of time required for information exchanges among OSS developers in **different** locations and the **same** locations respectively. This helps us more clearly distinguish between the time-lag by the geographical differences and the time-lag by the differences of developers' lifestyles.

**Appropriate timing for sending messages.** In order to identify the appropriate timing for communications which resolves communication time-lags as much as possible, the analysis method calculates the number of replied messages by each hour, using **posted (local) time** and **replied (local) time**. A numerical number in Fig.2(a) shows size of time-lag (hours) between time zones A and B. Fig.2(b) shows the number of pairs of posted messages from time zone A and replied messages from time zones B. For instance, suppose that one developer in A post a message between 9 and 12, and other developer in B replies a message



replied (local) time in time zones B

| posted (local) time in time zones A | 3–6 | 6–9 | 9–12 | 12–15 | 15–18 | 18–21 | 21–0 | 0–3 |
|---|---|---|---|---|---|---|---|---|
| 0–3 | 0 | +3 | +6 | +9 | +12 | +15 | +18 | +21 |
| 3–6 | +21 | 0 | +3 | +6 | +9 | +12 | +15 | +18 |
| 6–9 | +18 | +21 | 0 | +3 | +6 | +9 | +12 | +15 |
| 9–12 | +15 | +18 | +21 | 0 | +3 | +6 | +9 | +12 |
| 12–15 | +12 | +15 | +18 | +21 | 0 | +3 | +6 | +9 |
| 15–18 | +9 | +12 | +15 | +18 | +21 | 0 | +3 | +6 |
| 18–21 | +6 | +9 | +12 | +15 | +18 | +21 | 0 | +3 |
| 21–0 | +3 | +6 | +9 | +12 | +15 | +18 | +21 | 0 |

(a) size of time lag

replied (local) time in time zones B

| posted (local) time in time zones A | 3–6 | 6–9 | 9–12 | 12–15 | 15–18 | 18–21 | 21–0 | 0–3 |
|---|---|---|---|---|---|---|---|---|
| 0–3 | 7 | 17 | 16 | 2 | 5 | 3 | 2 | 0 |
| 3–6 | 0 | 9 | 7 | 5 | 3 | 2 | 2 | 0 |
| 6–9 | 0 | 0 | 9 | 13 | 9 | 8 | 4 | 1 |
| 9–12 | 0 | 2 | 2 | 61 | 80 | 24 | 12 | 2 |
| 12–15 | 1 | 8 | 5 | 6 | 57 | 35 | 17 | 0 |
| 15–18 | 6 | 13 | 11 | 3 | 4 | 54 | 57 | 1 |
| 18–21 | 4 | 39 | 11 | 3 | 0 | 1 | 30 | 12 |
| 21–0 | 4 | 42 | 7 | 3 | 7 | 2 | 2 | 6 |

(b) num. of replies

**Fig. 2.** Distribution of posted and replied time

between 15 and 18. In this case, the time-lag is +3 hours and the number of post/reply pairs is 80.

Time zones A and B are fixed after selecting target locations for analysis. Time zones B in Fig.2 is arranged as replied messages within an hour correspond to posted messages on the diagonal. In Fig.2, size of time-lag and the number of posted/replied messages are counted by three hours, but the length may be changed depends on analysis needs. Furthermore, the all cells in Fig.2(b) are gray-scaled according to the number of posted/replied pairs of messages, to grasp a big picture of time slots with a large or small number of replied messages.

Using Fig.2(a) and (b), it is possible to identify time slots with large or small time-lag. For instance, we can see that messages posted between 21 and 0 in time zones A (the bottom row in Fig.2) tend to be replied after 6 hours. That is, to post messages from 21 to 0 would not be the appropriate timing for less time-lag communications.

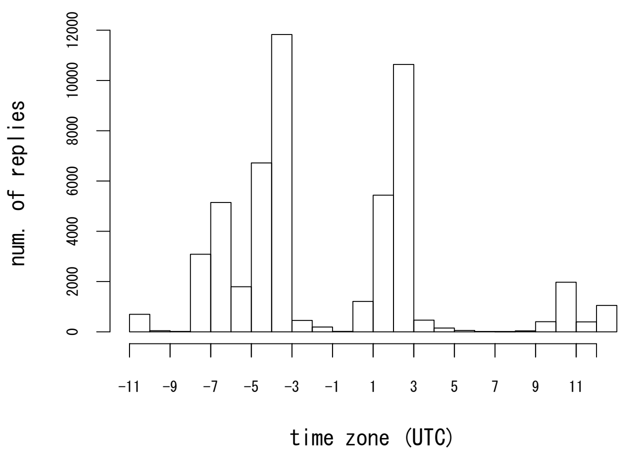


**Fig. 3.** Distribution of the number of replied messages by time zones

## 3 Case Study

This section describes a case study with a mailing list for developers in the Python project. Through the case study, we would like to confirm whether the analysis method can help us understand the existence of time-lags in communications among OSS developers.

### 3.1 Python

Python is an object oriented script language developed by OSS. It is very popular in Europe and the United States as well as Perl. Because it supports various platforms and provides rich documentations and libraries, it is used in a broad range of domains (e.g., Web programming, GUI-based appricaitons, CAD, 3D modeling, formula manipulation, and so forth).

### 3.2 Target data

We selected the mailing list archive called"Python-Dev" which is for discussing development of Python such as new features, release and maintenance. We use the Python-Dev mailing list archive from April 1999 to April 2009, which have 89,301 messages. Excluding posted messages with no replies and messages with no information on posted/replied time and locations, posted and replied messages were 56,707. 51,830 of 56,707 messages were sent within 24 hours.

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

**Table 1.** Target locations for the case study of Python

| region | time zone | locations |
|---|---|---|
| North and South American continent | UTC−8∼ UTC−4 | United States, Canada, West of Brazil, Chile, Bolivia, Mexico, etc. |
| European and African continent | UTC+0∼ UTC+3 | Europe, Africa, Moscow, Iran, Saudi Arabia, etc. |

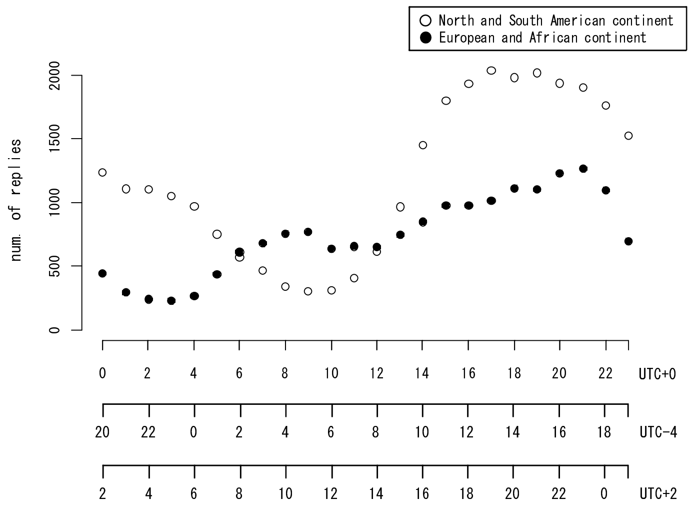


**Fig. 4.** Distribution of the number of replied messages by time slots (white circles: North and South American continent, black circles: European and African continent)

## 3.3 Analysis results

**Analysis of a geographical distribution and activity time.** Fig.3 shows a distribution of the number of replied messages by time zones. The X-axis and Y-axis respectively mean time zones and the number of replied messages.

Fig.3 indicates that in the Python project, a large number of messages are replied by developers from UTC−4 (East of the United States) and UTC+2 (central Europe). This result is not surprising at all. Because Python is mainly used and developed by European and American developers, it would be natural that developers living in the locations actively communicated.

Many of countries in the locations of UTC−4 and UTC+2 is utilizing daylight-saving time. And countries around the countries in UTC−4 and UTC+2 also have many messages. So, we selected two regions around UTC−4 (the North and South American continent: UTC−8∼UTC−4) and UTC+2 (the European and African continent: UTC+0∼UTC+3) as the analysis target in this paper. Table 1 shows major countries included in these regions.

**Table 2.** Statistics of time-lags by region (A: North and South American continent, E: European and African continent)

| posted region → replied region<br>replied region | the number of<br>replies | maximum<br>(hours) | median<br>(hours) | minimum<br>(hours) |
|---|---|---|---|---|
| A → A | 18,901 | 11.55 | 1.24 | 0.00 |
| A → E | 6,942 | 16.34 | 2.07 | 0.00 |
| E → E | 9,426 | 14.69 | 1.59 | 0.00 |
| E → A | 7,215 | 13.91 | 1.80 | 0.00 |

Fig.4 shows transitions of replied messages by hour in the two regions which are determined from Fig.3. The X-axes shows time in the three time zones (UTC+0, UTC−4, UTC+2) and the Y-axis is the number of replied messages.

Fig.4 indicates that the maximum and minimum number of replied messages from the North and South American continent are attained respectively at 13 and 5 in the local time (UTC−4). Python developers in the North and South American continent seem to mainly communicate during daytime hours. In contrast, Python developers in the European and African continent actively communicate during nighttime hours, because the number of replied messages from the European and African continent is peaked at 23 in the local time (UTC+2). In this way, analyzing activity time of OSS developers by using the number of replied messages helps us understand the existence of the difference of working hours by region.

Although Fig.4 provides an overview on the difference of working hours of OSS developers by region, however, it does not tell us anything about time-lags. In fact, developers in the both regions actively communicate each other from 12 to 23 in UTC+0. Communication time-lags might not exist in the regions. In contrast, developers in either one region or the other region does not actively communicate from 12 to 23 in UTC+0. Communication time-lags between developers living different locations might exist in this time period.

**Analysis of a distribution of time required for information exchanges among OSS developers in different locations.** Table 2 shows time spent to reply messages to the same and different time zones, the number of replied messages, and time-lags (maximum/median/minimum). A pair of a post from location X and a reply from location Y is represented as "X → Y".

The median hours of time-lag among the same time zone was 1.24 hours for A → A and 1.59 hours for E → E. The median hours of time-lag between the different time zones was 2.07 hours for A → E and 1.80 hours for E → A. Developers in the same time zone can expect to have a reply within 90 minutes, and developers between different time zones also can expect to have a reply within about 2 hours. Since the actual difference of time-lag between the target regions is nearly 6 hours, we can consider that communication time-lags in the Python project is relatively small.

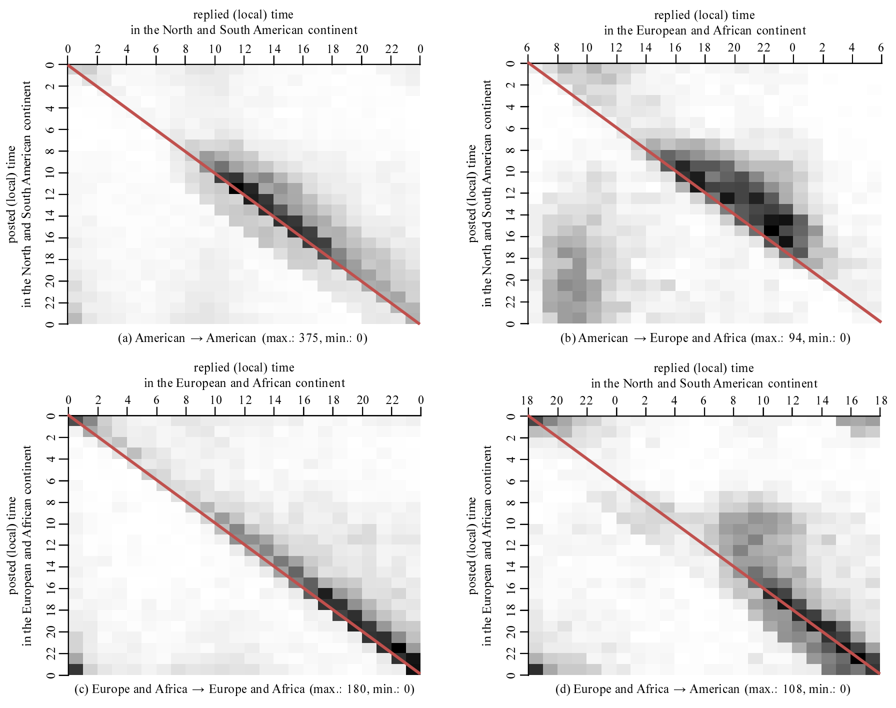


**Fig. 5.** Distributions of posted/replied local time between two regions

**Analysis of appropriate timing for sending messages.** Fig.5 (a), (b), (c) and (d) are distributions of the number of replied messages between two regions. For the simplicity, only gray-scaled figures without the number of replied messages are shown in Fig.5. We can see that the zero time-lag (i.e., dark gray cells near the diagonal line) is expected from 10 to 17 in Fig.5(a), from 9 to 17 in posted (local) time and from 15 to 23 in replied (local) time in Fig.5(b), from 16 to 23 in Fig.5(c), and from 16 to 23 in posted (local) time and from 10 to 17 in replied (local) time in Fig.5(d). For these time periods, developers would timely communicate each other.

In contrast, reply time seems to be delayed from 18 to 23 in posted (local) time in Fig.5(b) and from 7 to 13 in replied (local) time in Fig.5(d), because there are darker cells a short distance away from the diagonal line. These two posted (local) time periods correspond to the time period from midnight to early morning (0 to 6) in replied locations, which means that developers in replied locations was sleeping at the posted time.

From the result of Fig.5, in oder to receive a quick reply, it would be desirable to post a message from 10 to 17 in the North and South American continent, and from 16 to 23 in the European and African continent. On the contrary, it is not appropriate timing to post a message from 18 to 23 in the North and South American continent, and from 7 to 13 in the European and African continent,

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

since time-lag is likely to occur. In this way, our analysis method helps OSS developers know the appropriate timing so that they can resolve a time-lag of information exchange in an OSS project as much as possible.

## 4 Discussions

Opposite to what we expected before our case study, we have confirmed in Table 2 that the influence of time-lag due to the time zone difference was relatively small in the Python project. One reason of this phenomena might be that active time of Python developers is partly overlapping in the two regions. Although there are about 6 hours time-zone difference between the two regions, the active time in the North and South American continent was different from that in the European and African continent as shown in Fig.4. Therefore, active hours of Python developers in the two regions might overlap by coincidence from 10 to 17 in the North and South American continent (from 16 to 23 in the European and African continent). Another reason may be that the number of Python developers subscribed to the "Python-Dev" mailing list is sufficiently-large to quickly respond to a posted message at any time.

Our analysis method is not only useful in knowing the appropriate timing for communications among geographically-distributed OSS developers, but also useful in changing communication media used in an project. For instance, when a project replaces mailing lists with IRCs (Internet Relay Chat) as communication media, developers would be required to more precisely understand the appropriate timing for communications to resolve time-lag. In that case, our method would help developers know the better timing for real-time communications.

OSS developers are not necessary to be geographically-distributed, but they may be at the same region or location. Though our analysis method mainly aims to understand the communication time-lag arising from time-zone differences, it can be used for the time-lag due to lifestyle differences of OSS developers in the same region or location. OSS developers have no constraint on their working hours and they can freely engage in OSS development. At the same region, some developers can work in the morning and other developers can develop OSS at midnight. Depending on the differences of lifestyles of developers, time-lags could happen even if they live close to each other. In this situation, our method can provide an insight on the differences of active time in the same region and help developers understand the appropriate timing for sending messages.

The analysis method also can be used for distributed development in a company. Working hours in a company are fixed to some extent, but it is not necessarily that a developer in one site can communicate with other developers in another site at a particular time. In the prior study [3], time zone differences are visualized to understand and exploit overlapping hours in a distributed environment. Our method can not only visualize the time zone differences, but also allows developers to understand the easiness of communication at a particular time period, using the number of replied messages (i.e., density of working activity at a particular time period).

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

In this paper, we introduce the time-lag analysis method toward improving the communication efficiency of geographically-distributed OSS developers. The analysis method targets mailing list archive data as communication logs to reveal the existence of communication time-lags. Although IRC communications are often used in OSS projects and they can be our analysis target, communications using IRC do not work when developers one wishes to talk are off-line. So, IRC communication logs are not likely to well-capture communication time-lags.

In this paper, we have conducted a case study of the Python project, using the "Python-Dev" mailing list archive. Python-Dev consists of about 10 years mailing list archive data. So, it might be too large to show communication time-lags among Python developers at the fine-grained level. Actually, we have observed that communication time-lags in the Python project were relatively small. We suspect that this results from the size population of developers (subscribers) of Python-Dev. In Python-Dev[4], a posted messages must be read by a number of developers in the world and so it might be easy to have replies. In order to emphasize the existence of time-lags and its issues, in the near future, we need to analyze more specific situations such as the level of communications among module owners, reviewers and patch contributors.

## 5 Related Work

The issues on communication time-lag or delay in OSS development have been intensively studied in relation to bug modification processes with bug tracking systems in open source projects [5–15]. For instance, Wang et al. proposed several metrics to measure the evolution of open source software [14]. The metrics include the number of bugs in software, the number of modified bugs and so on. As a result of a case study using the Ubuntu project which is one of Linux-based operating system distributions, the study found that about 20% of all the reported bugs were actually resolved and over ten thousand bugs were not assigned to developers. These findings indicate that it takes a long time to resolve all bugs reported into bug tracking systems and that it also takes a long time to start modifying bugs. The study, however, did not reveal the amount of time or communication time-lags to resolve bugs.

Mockus et al. [12] and Herraiz et al. [7] have reported studies on the mean time to resolve bugs in open source software development. Mockus et al. [12] have conducted two case studies of the Apache and Mozilla projects to reveal success factors of open source software development. In the case studies, they analyzed the mean time to resolve bugs because rapid modifications of software bugs are generally demanded by users. As a result of the analysis, they have found that the mean time to resolve bugs were short if bugs existed in modules regarding to kernel and protocol, and existed in modules with widely-used functions. They also found that 50% of bugs with the priority P1 and P3 were resolved within 30 days, 50% of bugs with P2 were resolved within 80 days, and 50% of bugs with P4 and P5 were resolved within 1000 days. While [12, 7] mainly focused on precise understandings of bug modification processes in open source software development, we are interested in the influence of communication time-lags among developers on the bug modification process.

The issues on differences of time-zone and/or geographical distance in distributed development rather have been discussed in terms of the context of corporate (proprietary) software development [16–20]. For instance, Harbsleb et al. [18] have compared single-site development with milti-sites development and then revealed that development in the distributed environment introduced the delay of development speed. In contrast, Bird et al. [21] analyized the development of Windows Vista by comparing distributed teams with collocated teams from the aspect of the post-release failures of components. They have found a slight difference in failures, but the difference have been less significant. Nguyen et al. [22] also reported the similar phenomena in the Eclipse Jazz project. Although the lessons learned from these studies on distributed software development provides us a lot of useful insights, they are partly applicable to geographically-distributed OSS development due to the differences of lifestyles of developers even in the same region or location. In this paper, we tried to tackle this unique feature of time-lags in OSS development.

## 6 Conclusion and Future Work

In this paper, we proposed an analysis method for observing the time-lag of communications among developers in an OSS project and then facilitating effective communications. As the results of our case study applying the analysis method to the Python developers' mailing list archive, we could confirm that our analysis method helps geographically-distributed OSS developers understand that

- active time of developers are different from regions,
- communication time-lags in the Python project is relatively small, and
- there exists the appropriate timing for resolving communication time-lags as much as possible.

In this paper, our analysis method targets communication time-lags in the two regions with the time zone difference. In the future, we need to analyze regions and/or locations without time zone differences in order to better understand the influence of lifestyle differences of developers on communication time-lags. As described before, we still need to analyze more specific situations of time-lags at the fine-grained level.

## 7 Acknowledgment

This research is being conducted as a part of the Next Generation IT Program and Grant-in-aid for Young Scientists (B)–20700028, 21–8995, 20–9220 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

## References

1. Robles, G., Gonzalez-Barahona, J.M.: Geographic location of developers at sourceforge. In: Proceedings of the International Workshop on Mining Software Repositories. (2006) 144–150
2. Python Programming Language – Official Website: http://www.python.org/
3. Laredo, J.A., Ranjan, R.: Continuous improvement through iterative development in a multi-geography. In: 2008 IEEE International Conference on Global Software Engineering. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2008) 232–236
4. Python-Dev – Python core developers ML: http://mail.python.org/mailman/listinfo/python-dev
5. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering(FSE'08). (2008) 308–318
6. Godfrey, M.W., Tu, Q.: Evolution in open source software: A case study. In: Proceedings of the International Conference on Software Maintenance(ICSM'00). (2000) 131–142
7. Herraiz, I., German, D.M., Gonzalez-Barahona, J.M., Robles, G.: Towards a simplification of the bug report form in eclipse. In: Proceedings of the 2008 international working conference on Mining software repositories (MSR'08). (2008) 145–148
8. Ihara, A., Ohira, M., Matsumoto, K.i.: An analysis method for improving a bug modification process in open source software development. In: IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops, New York, NY, USA, ACM (2009) 135–144
9. Kim, S., Pan, K., Whitehead, Jr., E.E.J.: Memories of bug fixes. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering(FSE'06). (2006) 35–45
10. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic identification of bug-introducing changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering(ASE'06). (2006) 81–90
11. Kim, S., Whitehead, Jr., E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories(MSR'06). (2006) 173–174
12. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and mozilla. ACM Transactions on Software Engineering and Methodology **11**(3) (2002) 309–346
13. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 international workshop on Mining software repositories (MSR'05). (2005) 1–5
14. Wang, Y., Guo, D., Shi, H.: Measuring the evolution of open source software systems with their communities. SIGSOFT Softw. Eng. Notes **32**(6) (2007) 7
15. Yilmaz, C., Williams, C.: An automated model-based debugging approach. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. (2007) 174–183
16. Carmel, E.: Global software teams: collaborating across borders and time zones. Prentice Hall PTR, Upper Saddle River, NJ, USA (1999)

17. Karolak, D.W.: Global Software Development: Managing Virtual Teams and Environments. IEEE Computer Society Press, Los Alamitos, CA, USA (1999)
18. Herbsleb, J.D., Mockus, A., Finholt, T.A., Grinter, R.E.: An empirical study of global software development: distance and speed. In: Proceedings of the International Conference on Software Engineering. (2001) 81–90
19. Milewski, A.E., Tremaine, M., Egan, R., Zhang, S., Kobler, F., O'Sullivan, P.: Guidelines for effective bridging in global software engineering. In: Proceedings of the International Conference on Global Software Engineering. (2008) 23–32
20. Sangwan, R., Bass, M., Mullick, N., Paulish, D.J., Kazmeier, J.: Global Software Development Handbook (Auerbach Series on Applied Software Engineering Series). Auerbach Publications, Boston, MA, USA (2006)
21. Bird, C., Nagappan, N., Devanbu, P., Gall, H., Murphy, B.: Does distributed development affect software quality? an empirical case study of windows vista. In: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2009) 518–528
22. Nguyen, T., Wolf, T., Damian, D.: Global software development and delay: does distance still matter? In: Proceedings of the International Conference on Global Software Engineering. (2008) 45–54

# A Case Study on the Impact of Global Participation on Mailing Lists Communications of Open Source Projects

Ran Tang[1], Ahmed E. Hassan[2] and Ying Zou[1]

[1] Dept. of Elec. and Comp. Eng., Queen's University, Kingston, Ontario, Canada
{ran.tang, ying.zou}@queensu.ca
[2] School of Computing, Queen's University, Kingston, Ontario, Canada
ahmed@cs.queensu.ca

**Abstract.** Participants from different countries and across diverse time zones discuss important design decisions and resolve conflicts in open source projects using mailing lists. A good understanding of the social structure of these mailing lists and the impact of the global participant pool on that structure helps in managing these projects. In this paper, we present a case study which investigates the impact of global participation on communication on the developer mailing list for two large open source projects: PostgreSQL and GTK+. We find that a small group of participants from a limited number of countries dominate the mailing list while the rest of the participants contribute equally across all countries. We show that discussion threads are becoming more spread out across the globe over time. We also analyze the response delay for inquiries by newcomers to the mailing list. The delay in response to the initial inquiry depends on the country of the poster and the time when the message was posted to the list. Our findings shed light into the distribution and flow of knowledge about open source projects around the world.

**Keywords:** Global software development; Mining software repository

## 1 Introduction

Participants in an open source project heavily depend on mailing lists. These lists play a central and important role in facilitating communication among the globally distributed participants. Discussions on mailing lists often shape the future of the project and impact its progress. A good understanding of the social aspect of such discussions is needed. Prior research has demonstrated the impact of social structure on the technical structure of large software systems (e.g., [3]).

Given the global nature of open source projects, we wish to explore the social structure in the context of the global pool of the participants. In particular, we examine the interaction and communication of participants from different countries and diverse time zones on the mailing lists of open source projects. Such study helps shed light into global software development practices, and would be of great help for managers working on distributing projects across a global pool of developers [4].

In prior work [22], we developed a technique to determine the country of a mailing list participant. Using this technique we can analyze the participation and interaction patterns on mailing lists. Our technique can determine the country of 67% of participants on a mailing list. This represents an 80% improvement over prior techniques (e.g.,[8]). Using our new technique we study the impact of global participation on mailing list communications. Our findings are derived from mining a total of 20 years of the mailing list repository for two large and long-lived open source projects: The PostgreSQL and GTK+ projects. Our contributions are centered along the following three questions:

1. **What is the participation rate of countries on the mailing list?** We study whether the mailing discussions are used mostly by participants centered in a limited number of countries or distributed across the globe. We show that a small group of participants from a small number of countries dominates the list while the rest of the participants contribute equally across all countries.
2. **How global are discussions on the mailing list?** We examine if discussions show a bias to being local, i.e., given a particular discussion whether there is a tendency for participants from close by regions to participate or if the participation pattern is more global. We find that most discussion threads are well spread across the globe and that the average spread of threads increases over time. This indicates that discussions span the globe instead of being limited to specific regions.
3. **How are inquiries by newcomers handled?** We study the speed and rate of responding to inquiries by newcomers. We find that the speed and rate of responding to inquiries by newcomers depend on the country of the email sender and the time that the email was posted to the mailing list.

**Organization of the paper.** The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents our research questions and the result of our case study. Section 4 discusses the threats to validity of our work. Finally, Section 5 concludes the paper.

## 2 Related Work

Prior studies of the global pool of participants in open source projects can be categorized into two groups: ones based on surveys and others based on mining repositories.

**Studies based on surveys.** Robles et al. [14] surveyed over 5,500 respondents to identify their country. Robles et al. showed that a majority of open source developers are from Europe. Similar results are also reported in the survey conducted by Ghosh [7] and David et al. [5].

**Studies based on mining repositories.** Dempsey et al. [6] analyzed the top-level domain name of the email address (e.g., .ca, .com) of a participant to identify the country of the participant. However the study did not compensate for the US bias resulting from the wide use of generic domains (e.g., .com). Prior techniques do not map .com addresses to any country. Therefore, the participants from the US may be under represented in the analysis. Studies in [12, 14] show that the developer pool is becoming more European-based over time. Robles and Gonzalez-Barahona [8, 13] used a technique to identify countries of participants in SourceForge [21] open source projects. Email address and time zone information in the user profile are analyzed to infer the country. The mailing list was also studied using a similar technique. However, the time zone information in the mailing list does not contain specific country information. Therefore, the analysis of the time zone can only derive the origins of participants to specific time zone regions instead of particular countries.

**Other work on mining mailing list.** Several studies mine mailing list repositories. For example, Mockus et al. [16] conduct two case studies to reveal the process of open source development using mailing list repository. Bird et al. [1] build social networks using information derived from the PostgreSQL mailing list. These studies do not explore the impact of geographical distribution on the social interaction of participants.

## 3 Case Study

We conducted a case study to explore various aspects of participation and interaction in the mailing lists. We use the developer mailing lists for the PostgreSQL [20] (postgresql-hackers) and GTK+ [9] (gtk-devel-list) projects in our case study.

**Table 1.** Statistics about the studied mailing lists

| | Studied Period | # of Participants | # of Threads |
|---|---|---|---|
| **PostgreSQL** | 1999-2008 | 4,742 | 23,104 |
| **GTK+** | 1999-2008 | 2,734 | 7,481 |

**Table 2.** Research questions

| | Research Questions |
|---|---|
| **Q1** | What is the participation rate of countries on the mailing list? |
| **Q2** | How global are discussions on the mailing list? |
| **Q3** | How are inquiries by newcomers handled? |

The PostgreSQL project is a relational database management system. The GTK+ project is a toolkit for creating cross platform graphical user interfaces. Both projects involve a large pool of international developers who interact through the mailing lists. Both projects come from two different domains: database management and graphic user interface development. Our objective is to study if our results hold across domains and projects. Table 1 presents descriptive statistics about both projects. Using the recovered countries for the participants, we sought to explore the research questions listed in Table 2. For each question, we present our motivation, and discuss our results using data from the mailing list repositories in the PostgreSQL and GTK+ projects.

## 4.1 What is the participation rate of countries on the mailing list?

**Motivation.** Prior research shows that open source projects have a small core team and a small number of core contributors [16, 19]. We want to identify the core participants in the mailing lists and study their distribution around the world. We wish to compare the participation of that small core with the rest of the participants. In particular, we want to examine if they are localized to small number of countries or if they are distributed around the globe. Such knowledge would be helpful in the planning and recruiting processes for open source projects. For example, the knowledge of global distribution and involvement of participants may help conference planner select an optimal locations for face-to-face project conferences (e.g.: PostgreSQL Conference [18]) in order to achieve high attendance.

**Results.** We measure the number of participants who contribute the majority (i.e., 70%) of the messages to the mailing list. We call these participants the core participants (similar to [16]). Table 3 shows that although these core participants represent a small percentage (i.e., 1.5-5%) of all the participants, they are spread out over a relatively larger percentage of countries.

**Table 3.** Statistics of core participants

| | **#participants (%participants)** | **# countries (%countries)** |
|---|---|---|
| **PostgreSQL** | 47 (1.5%) | 13 (13%) |
| **GTK+** | 96 (5%) | 21 (27%) |

Table 4 and Table 5 show in more detail the countries of all participants. We observe that the participation patterns vary between projects. For instance, although the US has the highest number of participants in both projects, the number of messages sent by US participants varies considerably in both projects. While the US participants contributing most (~58%) of the messages on the PostgreSQL mailing list, they only contribute ~20% of the messages on the GTK+ mailing list with Germany being the top contributor of messages.

Table 6 and Table 7 demonstrate an interesting pattern for the contribution of countries to the mailing list. Looking at the median of the number of messages and threads in each country, we find that the median is surprisingly very low and that it is consistent across countries. The majority of participants post 1 or 2 messages and are involved in 1 or 2 threads. This pattern leads us to hypothesize that most participants

**Table 4.** Country composition of the PostgreSQL mailing list

| Country | Participants (%) | Msgs (%) |
|---|---|---|
| United States | 1037(32.6%) | 76723(57.8%) |
| Germany | 228(7.2%) | 7237(5.5%) |
| Canada | 160(5.0%) | 7602(5.7%) |
| UK | 144(4.5%) | 8584(6.5%) |
| Australia | 108(3.4%) | 4862(3.7%) |
| Russia | 98(3.1%) | 2578(1.9%) |
| India | 97(3.0%) | 574(0.4%) |
| France | 97(3.0%) | 1621(1.2%) |
| Italy | 92(2.9%) | 424(0.3%) |
| Brazil | 90(2.8%) | 424(0.3%) |
| Japan | 89(2.8%) | 3979(3.0%) |
| Netherlands | 66(2.1%) | 722(0.5%) |
| China | 54(1.7%) | 210(0.2%) |
| Poland | 51(1.6%) | 326(0.2%) |
| Czech | 48(1.5%) | 940(0.7%) |
| Austria | 47(1.5%) | 3247(2.5%) |
| Sweden | 44(1.4%) | 2974(2.2%) |
| Hungary | 41(1.3%) | 271(0.2%) |
| Spain | 37(1.2%) | 227(0.2%) |
| Denmark | 28(0.9%) | 209(0.2%) |
| New Zealand | 28(0.9%) | 1,024(0.8%) |
| Other | 492(15.5%) | 7,891(6.0%) |

**Table 5.** Country composition of the GTK+ mailing list

| Country | Participants (%) | Msgs (%) |
|---|---|---|
| United States | 517(27.8%) | 4,623(19.9%) |
| Germany | 189(10.2%) | 6,670(28.7%) |
| France | 124(6.7%) | 1026(4.4%) |
| UK | 120(6.5%) | 3,111(13.4%) |
| Sweden | 64(3.4%) | 800(3.4%) |
| Australia | 63(3.4%) | 708(3.0%) |
| Canada | 57(3.1%) | 429(1.8%) |
| Italy | 55(3.0%) | 260(1.1%) |
| India | 53(2.9%) | 173(0.7%) |
| Netherlands | 50(2.7%) | 268(1.1%) |
| Spain | 42(2.3%) | 216(0.9%) |
| China | 41(2.2%) | 1469(6.3%) |
| Finland | 32(1.7%) | 864(3.7%) |
| Russia | 29(1.6%) | 242(1.0%) |
| Brazil | 27(1.5%) | 147(0.6%) |
| Japan | 25(1.3%) | 110(0.5%) |
| Austria | 23(1.2%) | 65(0.3%) |
| Belgium | 23(1.2%) | 134(0.6%) |
| Czech | 22(1.2%) | 113(0.5%) |
| Norway | 21(1.1%) | 99(0.4%) |
| Other | 283(15.2%) | 1,733(7.5%) |

**Table 6.** Participation level for the PostgreSQL mailing list

| Country | Mean Msgs | Med Msgs | Mean Threads | Med Threads |
|---|---|---|---|---|
| US | 74 | 2 | 32.9 | 1 |
| Germany | 31.7 | 2 | 17.3 | 1 |
| Canada | 47.5 | 2 | 24.3 | 2 |
| UK | 59.6 | 3 | 27.6 | 2 |
| Australia | 45 | 2 | 24.8 | 1 |
| Russia | 26.3 | 2 | 14.4 | 1 |
| India | 5.9 | 1 | 3.5 | 1 |
| France | 16.7 | 2 | 8.7 | 1 |
| Italy | 4.6 | 2 | 3 | 1 |
| Brazil | 4.7 | 2 | 3.1 | 1 |
| Japan | 44.7 | 2 | 19. | 2 |
| Netherlands | 10.9 | 2 | 6.3 | 2 |
| China | 3.9 | 2 | 3 | 1 |
| Poland | 6.4 | 2 | 3.4 | 1 |
| Czech | 19.6 | 3 | 11.1 | 2 |
| Austria | 69.1 | 4 | 42.7 | 2 |
| Sweden | 67.6 | 2.5 | 30.1 | 1.5 |
| Hungary | 6.6 | 2 | 3.4 | 1 |
| Spain | 6.1 | 1 | 3.3 | 1 |
| Denmark | 7.5 | 3.5 | 5.2 | 2 |
| New Zealand | 36.6 | 2.5 | 19.2 | 1.5 |

**Table 7.** Participation level for the GTK+ mailing list

| Country | Mean Msgs | Med Msgs | Mean Threads | Med Threads |
|---|---|---|---|---|
| US | 8.9 | 2 | 5.8 | 2 |
| Germany | 35.3 | 2 | 21.7 | 2 |
| France | 8.3 | 2 | 5.6 | 1 |
| UK | 25.9 | 2 | 15.2 | 2 |
| Sweden | 12.5 | 2.5 | 8.4 | 2 |
| Australia | 11.2 | 2 | 8.5 | 1 |
| Canada | 7.5 | 2 | 4.3 | 1 |
| Italy | 4.7 | 2 | 2.8 | 1 |
| India | 3.3 | 1 | 2.7 | 1 |
| Netherlands | 5.4 | 1 | 3.6 | 1 |
| Spain | 5.1 | 2 | 3.6 | 1 |
| China | 35.8 | 1 | 21.3 | 1 |
| Finland | 27 | 2.5 | 16.6 | 1.5 |
| Russia | 8.3 | 2 | 4.8 | 1 |
| Brazil | 5.4 | 2 | 3.1 | 2 |
| Japan | 4.4 | 2 | 2.5 | 1 |
| Austria | 2.8 | 1 | 2.1 | 1 |
| Belgium | 5.8 | 2 | 3.6 | 1 |
| Czech | 5.1 | 2 | 3.6 | 1 |
| Norway | 4.7 | 1 | 2.3 | 1 |

rarely use the mailing list for discussion. Instead they post 1 or 2 inquiries in the mailing list. In short, most mailing list participants use it to post inquiries, rather than to delve into in-depth discussions.

These observations about the different use of mailing lists by core members and newcomers shape our next two questions. Q2 will study the spread of countries in in-depth discussion threads while Q3 will study the speed of response to inquiries by newcomers.

Developer mailing lists are dominated by a very small number of participants who are from a relatively larger number of countries. Participants contribute equally to the mailing lists independent of their country.

## 4.2 How global are discussions on the mailing list?

**Motivation.** The results for our previous question indicate that the participant pools in both projects are globally distributed. However, how these participants interact remains unanswered. Do participants talk globally or do they prefer to talk to participants locally? This question examines the interaction in the global open source development. This helps us gain insight about the problems of coordination and localization of knowledge for distributed teams.

**Results.** We define a spread metric for each thread to measure the global spread of a thread. A discussion thread is a collection of email messages related to each other by replying. A participant starts a discussion thread by posting a question or raising an issue, and other participants may choose to reply to it. By examining the diversity of the participants' locations in each thread, we can determine whether the discussion is global or primarily localized.

We use the MESSAGE-ID field in an email to reconstruct discussion threads since each message, as part of a thread, would refer to the message id of an earlier message in the thread. Sometimes, a thread is re-opened for some reason. For example, participants may reply to a thread which has had no postings for more than one year. Since the discussion has stopped for too long, this reply essentially creates a new discussion on the same topic. We process such a reply as the starting point of a new thread if the time between the reply and the last posting is a long period of time. We use a threshold of 30 days to cut off re-opened threads into two different threads. This threshold is selected by manually examining the re-opened threads in both studied projects.

To compute the spread of a thread, we calculate the spread between each pair of participants who have posted on the thread. As shown in Figure 1, for each pair of participants A and B, we use either the time zone difference from A to B (clockwise) or B to A (clockwise), whichever is less, as the spread between them. The maximum spread between two participants is 12. The spread is 0 if both participants are in the same time zone. In the example shown in Figure 1, the spread is 8. We pick the largest spread between all pairs of participants in a thread. A large spread is a good indicator of the global spread of interaction in a thread.

We consider threads with a spread less or equal to 5 as low spread threads. Typically, such threads (i.e., with the spread of 5) represent discussion within one country (e.g., US) or one close region (e.g., EU). We then examine the trend of low spread threads relative to all threads over time.

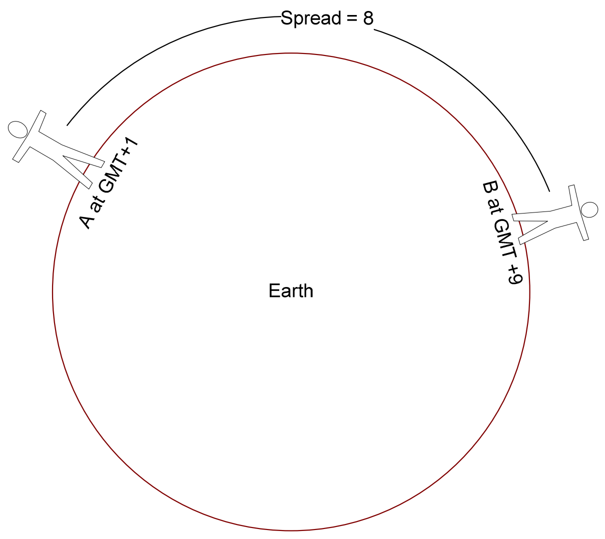


**Figure 1: An example of spread calculation**

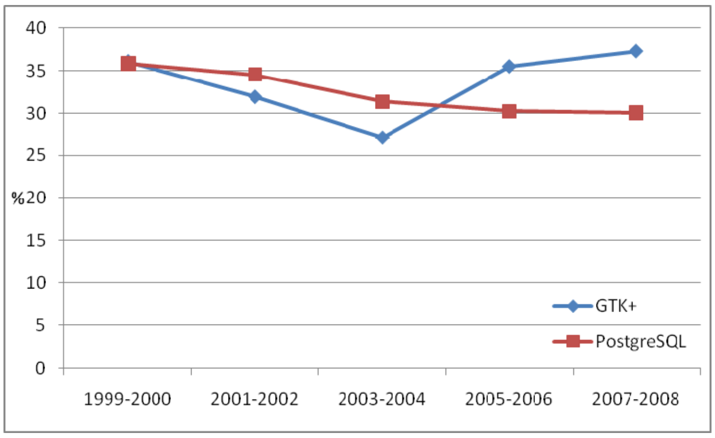


**Figure 2:** % of low-spread threads over time

As shown in Figure 2, low spread threads only represent about one third of the total threads and low spread threads decrease in the PostgreSQL project over time. The participants of PostgreSQL list were primarily from the US when it was started in

1999. The mailing list attracts more participants throughout the world and the PostgreSQL project is becoming more international over time.

The GTK+ project illustrates another interesting trend. In early years, it follows a similar trend with the project being localized in the US. We then note that the percentage of low-spread threads continues to decrease. This project was at first centralized in the US, and then it grew to have European participants till 2003. However, this trend is reversed with more localized European threads since 2003. An observation about the project population of GTK+ and other GNOME projects was noted by [12] which mined the source code change logs of the credit, instead of examining the mailing lists participants. The confirmation of our mailing list findings by mining the source code repositories demonstrates the importance of social information in explaining and collaborating information recovered from other project repositories.

Discussion threads tend to become more global as a project evolves. However, this trend might change over time for some projects.

### 4.3 How are inquiries by newcomers handled?

**Motivation.** Prior research [13] and our earlier results show that many open source projects have a high concentration of participants from US, Canada or European Union (EU). We believe that the high concentration of participants might impact the openness of such projects. The openness of the mailing list is an important factor that influences newcomers. For example, if inquiries by newcomers are often ignored or take a long time to get a response, then newcomers might lose their interest in the project and not join the community [2]. Two factors which might affect the delay in responding to an initial inquiry are the country of the poster and the posting time (which might be indirectly affected by the country). For example, posts by Chinese participants might not get an immediate response till next day when the North American participants are at work.

**Table 8.** Response delay and ratio for both studied projects

| | EU, US, Canada | | Others | |
|---|---|---|---|---|
| | **Response Delay (hours)** | **Response Ratio (%)** | **Response Delay (hours)** | **Response Ratio (%)** |
| **PostgreSQL** | 0.17 | 68.2% | 1.66 | 67.0% |
| **GTK+** | 0.46 | 61.9% | 19.7 | 59.0% |

**Results.** In our analysis, we divide all countries into two groups:

1. Group one includes countries from the EU, the US and Canada. This group includes many developed countries which are known to be active in open source development [8].

2. Group two includes all other countries. This group mostly includes developing countries with less open source development activity.

We measure the median response delay and the response ratio for each group. We define the response delay for an initial inquiry as the time difference between the initial inquiry and the first reply. The response ratio is denoted as the number of replied inquiries divided by the total number of inquiries throughout the studied period for each project. We only examine participants who have less than 20 messages in the mailing list. We choose not to consider core participants with more messages since we believe that the rest of the mailing list is familiar with them and that they would receive a response independent of their country or time of posting.

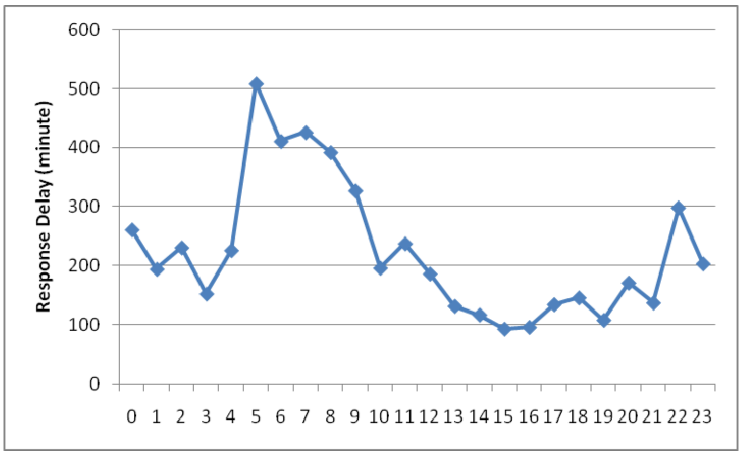


**Figure 3.** Response delay for the PostgreSQL project (GMT)

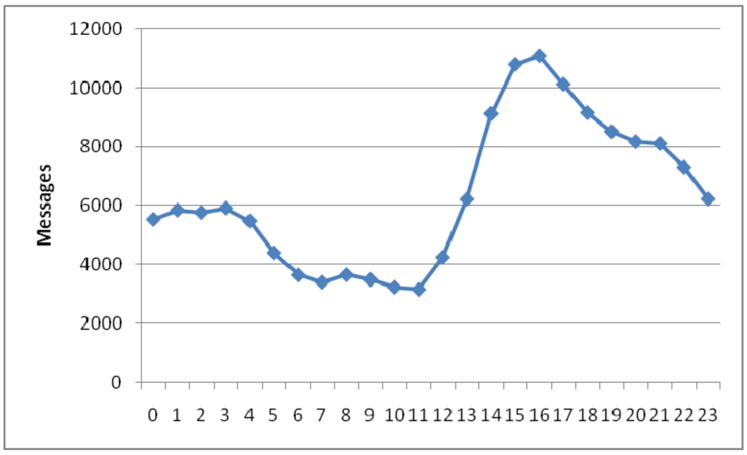


**Figure 4.** Traffic on the PostgreSQL mailing list (GMT)

As shown in Table 8, newcomers from the EU, the US and Canada get a statistically significant faster response compared to those from other countries. As for the response ratio, the ratios are consistent across both groups with very low variations that are not statistically significant. The response delay and ratio for the PostgreSQL project are much lower than the GTK+ project. One possible reason is the fact that the PostgreSQL project has a larger participant pool which is more distributed as shown in Q2.

We plotted the response delay for both projects throughout a day. Figure 3 depicts the response delay for the newcomers in the PostgreSQL project. There is a large spike in delay between 5 to 11 GMT. In an effort to investigate this spike in delay, we plotted the traffic (i.e., the total number of messages posted per GMT hour of a day) in the PostgreSQL mailing list shown in Figure 4. The patterns in Figure 3 and Figure 4 are inverted. To better quantify the relation, we calculated the spearman correlation between both metrics plotted in Figure 3 and Figure 4. The correlation is -0.79 for the PostgreSQL project and -0.59 for the GTK+ project, indicating a strong negative correlation. In short, when the traffic is high, the response delay is low; and vice versa. Although one might assume that an open source mailing list provides around the clock support, the mailing list in many ways operates as a traditional company which has specific support hours and reduces staff in the off-peak hours. For most participants, it is of little value to post a message in off-hours, since there is a high chance that the message won't receive a reply till the list is active again (probably the following day).

Our analysis shows that the response delay depends on the country and the posting time over the participants from all countries. We sought to explore the response delay in a particular country. We picked the US with a sufficient number of inquiries that can be spread over 24 hours. We studied its response delay pattern. As depicted in Figure 5, the response delay pattern is similar to the pattern described in the overall graph (Figure 3) for the PostgreSQL project. Therefore, we believe that the response delay for an active country also depends on the posting time.

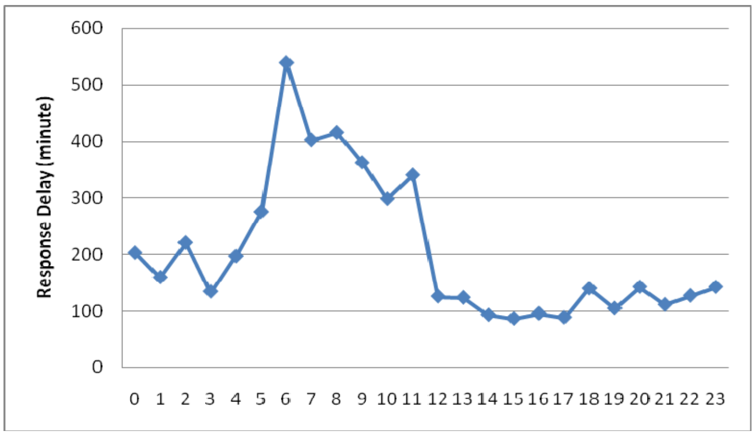


**Figure 5.** Response delay over one day for US based participants for the PostgreSQL

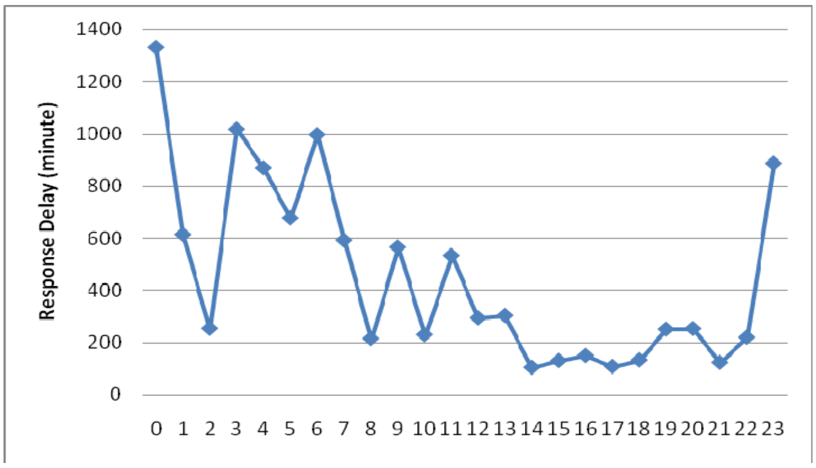


**Figure 6.** Response delay over one day for US based participants for GTK+

The response delay and ratio to a newcomer's initial inquiry depend on the time when the inquiry was posted and on the country of the participant who posted it. Open source mailing lists do not operate at full capacity (e.g., 24 hours over 7 days). Instead the open source mailing lists operate in the similar way as support lines with high and low staffing periods based on the time of a day.

## 5 Threats to validity

Our work has several limitations which affect the validity and the generality of our findings. First, our findings are based on studying two open source projects. We chose two long-lived successful and active open source projects for our study with a well-archived mailing list repository. In future work, we need to explore additional projects to verify the generality of our findings. It would be interesting to explore non-successful projects though the mailing lists of such projects are not as active and likely won't have as much discussions.

Our approach [22] to identify the location of a participant use IP2Location databases [11]. These databases are built using several heuristics and might contain errors [17]. Moreover, it might be the case that the location of a specific IP has changed over time with the IP2Location database mapping an out-of-dated IP address to the most recent location recorded in the database. Multi-national companies might have their whole intranet accessing the internet through US-based gateways. This would cause all remote offices to appear as if they are in the US. We determine the location of a participant based on the most frequently reported country using the sender IP address analysis. However, a participant may move from one country to another over the years. We also assume that each participant can reside in a single country. A cursory analysis of the data shows that almost all participants have 90% of their posts coming from the same country. Nevertheless, we plan to explore these assumptions in future work.

A final limitation of our approach is that the correct country of a participant is not known. In other words, there exists no gold standard to compare against. As a basic accuracy verification of our approach, we compared the identified participants by both our approach and prior work [8]. We found that both approaches have a 4-5% mismatch ratio in identified countries. Studying the mismatches, we find that 70-80% of them are due to our approach using the most frequent IP location of the sender instead of mapping a participant to the country indicated by resalable country domain name.

We have defined a few thresholds, such as the low spread threads (less than 5 time zones) and the intervals for creating a new thread from an inactive thread (i.e., 30 days). These thresholds work well in analyzing the two studied projects. In the future, we plan to examine other possible thresholds on more projects.

Much of our findings show correlation between attributes without explaining the causes. More studies are needed to explore the causes. For example, our findings show that the initial inquiries submitted by the participants from the regions outside of the US, Canada, and the EU have a high response delay. However, we need to further explore the reasons through ethnographical studies.

## 6 Conclusion

Studying communication on mailing lists shed light into the spread and flow of knowledge for a project. Through a case study on two large and long-lived open source projects: PostgreSQL and GTK+, we investigated the impact of having globally distributed participants communicating on the mailing list. We found that a small number of participants spread over a larger set of countries dominate the discussion. We found that the majority of participants contribute a single message to a single thread. We noted and examined two different uses of the mailing list: a) for lengthy discussions by core members; b) for inquiries by newcomers. For lengthy discussions, we found that over time the discussions become more spread out across the globe for one of the studied projects, while the other project (GTK+) has the same trend in the beginning but later becomes less spread out. A closer analysis indicates that mailing list participants often reflect the developer composition of a project. In the case of the GTK+ project, our analysis noted the migration of the development team from the US to the EU. As for inquiries by newcomers, we found that delays in responding to such inquiries depend on the country of the newcomer and the posting time of the inquiry. Our results help us better understand the social structure and global nature of open source projects, and their impact on timely and open discussions in open source projects.

## References

1. Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. Mining email social networks in Postgres. International Workshop on Mining Software Repositories, 2006.
2. Bird, C., Gourley, A., Devanbu, P., Swaminathan, A., Hsu, G. Open Borders? Immigration in Open Source Projects, International Workshop on Mining Software Repositories, 2007.
3. Bowman, I. T. and Holt, R. C. Software architecture recovery using Conway's law. Conference of the Centre For Advanced Studies on Collaborative Research, 1998.
4. Cherry, S., Robillard, P. N. Communication Problems in Global Software Development: Spotlight on a New Field of Investigation. International Workshop on Global Software Development, 2004.
5. David, P.A., Waterman, A., Arora, S. FLOSS-US. The free/libre/open source software survey for 2003. Technical Report, Stanford Institute for Economic and Policy Research, 2003.
6. Dempsey, B.J., Weiss, D., Jones, P., Greenberg, J. Who is an open source software developer? Communications of the ACM, 45(2), 2002.
7. Ghosh, R.A., Glott, R., Krieger, B., Robles, G. Survey of developers (free/libre and open source software: survey and study). Technical Report, University of Maastricht, 2002.
8. Gonzalez-Barahona, J. M., Robles, G., Andradas-Izquierdo, R. and Ghosh, R. A., Geographic origin of libre software participants, Information Economics and Policy, 20 (4), December 2008.
9. GTK+ developer mailing list, http://mail.gnome.org/mailman/listinfo/gtk-devel-list/, last accessed on October 29 2009.
10. Hertel, G., Niedner, S., Herrmann, S. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. Research Policy, Volume 32, 2003.
11. IP2Location Database, http://www.ip2location.com/, last accessed on October 29 2009.
12. Lancashire, D. Code, culture and cash: the fading altruism of open source development. First Monday, 6 (12), December 2001.
13. Robles, G., Gonzalez-Barahona, J. M. Geographic location of developers at SourceForge. International Workshop on Mining Software Repositories, 2006, pp. 144-150.
14. Robles, G., Scheider, H., Tretkowski, I., Weber, N. Who is doing it? A research on libre software developers. Technical Report, Technische Universitt, 2001.
15. Member states of the EU, http://europa.eu/abc/european_countries/index_en.htm, last accessed on October 29 2009.
16. Mockus, A., Two case studies of open source software development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology, 11(3), Jul. 2002.
17. Padmanabhan, V., Subramanian, L. An investigation of geographic mapping techniques for Internet hosts. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2001.
18. PostgreSQL conference, http://postgresqlconference.org/ last accessed on October 29 2009.
19. PostgreSQL contributor profiles, http://www.postgresql.org/community/contributors/, last accessed on October 29 2009.
20. PostgreSQL developer mailing list, http://archives.postgresql.org/pgsql-hackers/, last accessed on October 29 2009.
21. SourceForge open source project repository, http://sourceforge.net, last accessed on October 29 2009.
22. Tang, R., Hassan, A.E., Zou, Y., Techniques for Identifying the Country Origin of Mailing List Participants, 16th Working Conference on Reverse Engineering, 2009
23. Tuomi, I. Evolution of the Linux credits file: methodological challenges and reference data for open source research. First Monday, 9(6), June 2004.

## Keynote address:

## Understanding Networked Collaboration

***Shuichiro Yamamoto (NTT Data Corporation, Japan)***

**ABSTRACT**

My recent interest is the mutual interaction among people and IT systems in different research domains. These domains include Knowledge creation, Requirements Engineering, and Dependable Systems Engineering. In this talk, I will propose several ideas to understand the mutual interactions among actors, where both people and IT systems are treated as actors. These ideas are Intermediary Knowledge, Actor relationship matrix and Engineering case pattern language. I am also planning in the future to integrate these approaches for designing Knowledge Collaboration through computer networks.

# Identifying the concepts that are searchable with keywords in code search engines

Toshihiro Kamiya

National Institute of Advanced Industrial Science and Technology
Akihabara Dai Bldg. 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 101-0021, JAPAN
t-kamiya@aist.go.jp

## 1 Introduction

Many code search engines, such as Codase (www.codase.com), Codefetch (www. codefetch.com), Google code search (google.com/codesearch), JExamples (www. jexamples.com), Koders (www.koders.com), Krugle (www.krugle.org), and Merobase (www.merobase.com), have become available recently [1-6]. Most of them have Google-like interfaces through which a user can enter a set of keywords as a query to retrieve source code files that are related to the keywords in the query. Some code search engines also provide options that are specific to source code. For example, software developers can use options to specify the specific portions (such as comments, code, or functional definitions) in which the search keywords appear.

Such code search engines are important instruments to promote and support software reuse. However, their support for reuse may not be sufficient. When software developers consider reuse, they care about not only the functionality of the code, but also various characteristics such as performance ("Is the algorithm O($N$) or O($N^2$)?"), usability ("Whether the API is easy to understand and use?"), and maintainability ("Can the code be easily customized to fit my code?"). This paper tries to evaluate the capabilities of keyword-based code search engines in terms of their support for searching reusable code based on multiple characteristics. The paper adopts what we call an oracle approach for the evaluation: it first identifies a classification schema that represents different dimensions of code characteristics, and then analyzes whether we are able to identify, for each dimension of characteristic, a set of intuitive keywords that can be used in a search query to retrieve effectively reusable code.

## 2 The Oracle Approach

We describe the oracle approach using a case study of searching code in the following scenario. A developer is writing a Java program that needs an array of bits with low memory consumption, that is, a class of bit array that uses one bit in the memory for each element. The developer guesses, by analogy of the class Array of the standard Java library, that the name of such a class may be called BitArray. So the developer can use BitArray as the functionality query for searching. However, the

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

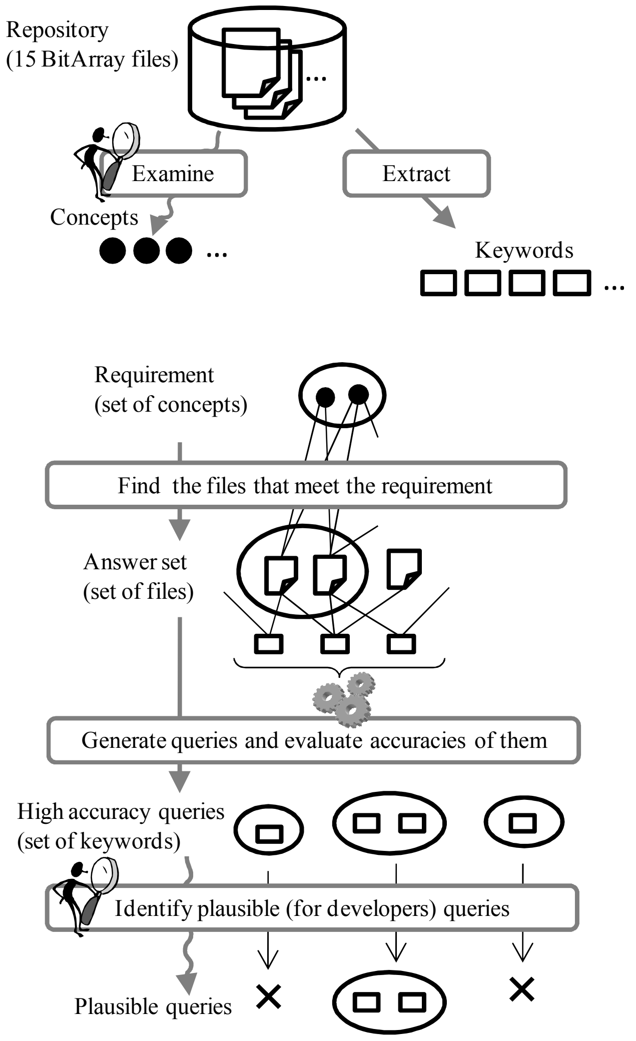


**Fig. 1.** Steps of the oracle approach

developer also has other requirements such as performance, comprehensibility and maintainability, and the question is what kind of keywords that he or she should use to represent such requirements for the purpose of searching.

To evaluate the oracle approach of finding effective search keywords that capture the requirements of multiple characteristics, we will use a toy keyword based code search engine that is prepared for this evaluation. The oracle approach (Fig. 1) of finding highly discriminative search keywords works as follows. For each predefined "correct" answer set of desired code that we want to find, we create a series of keyword sets that are used as search queries. Each query will return a set of search results, the search results are then measured against the predefined correct answer set.

Based on the measurement, we will find whether keywords with high discriminating power exist. More specifically, the approach consists of the following steps:

(1) Prepare a repository of source files, which are the candidates for code search.
(2) Examine each source file in the prepared repository and create a list of *concepts* that can be used to describe various features of source files, including basic functionality and other implementation details such as performance and potential usage pitfalls. This step needs to be performed by a subject expert.
(3) Extract a list of keywords from each source file to represent the source file.
(4) From the list of concepts, create a classification schema by putting each concept into different categories. This classification schema represents different dimensions of search *requirements*. A set of search requirements is created, and each search requirement contains a subset of the concepts, and in this case study, one search requirement contains one concept from each category.
(5) For each requirement,
  (5-1) Create an *answer set*, which is a set of source files from the repository that contain the concepts of the requirement.
  (5-2) Determine what words can make a query that returns search results with high accuracy, namely, identifying the words that have the highest discriminative power in terms of search accuracy. To do this, we create search queries with arbitrarily selected keywords from the keyword lists, and then compare the search results of those queries with the predefined *answer set*.
  (5-3) For each query that achieves high search accuracy, analyze whether it is plausible for a software developer to include such words in their search queries based on their search requirements. The search accuracy of a query is evaluated with precision and recall.

## 3 A Case Study

### 3.1 Source files, concepts, and keywords

The repository in the case study contains 15 Java source files of different implementations of the BitArray class that are found with existing code search engines. Table 1 shows the concepts that are identified by analyzing the source files. The concepts are classified into 5 categories: *basic operation, scale, implementation issue, rich operation, conversion, and ease of development*. The concepts of limited-size, unlimited-size and re-size are mutually exclusive; that is, a source file can have only one concept from that category. Non-pack and pack in the implementation category are also mutually exclusive.

Keyword lists were generated from the 15 Java source files with a small script. Camel cased identifiers such as `getLength` are split into separate words (e.g. "get" and "length"). Tag names (e.g. `@author`) in JavaDoc comments were removed. Operators (such as <<=) were extracted as words too.

**Table 1.** Concepts extracted from the source files.

| Concept | Classi-fication | Description |
|---|---|---|
| basic | basic op. | Can store bits and retrieve the stored bits. |
| limited-size | scale | The max count of bits are hard-corded in a source file (a constat). |
| unlimited-size | | Size of a bit array is specified on instance creation. |
| re-size | | Size of a bit array can be modified with methods. |
| non-pack | impl. issue | One byte or more stroage is required to store a bit. |
| pack | | Less than a byte is required to stroe a bit. |
| mask | | A workaround to prevent a time-consuming micro operation. |
| break-encap. | | Has methods that return internal data (stored bits). |
| search | rich op. | Has methods to search true bits in a bit array. |
| merge | | Has methods to merge two bit arrays. |
| value | | Has a predicate for equality/comparision between two bit arrays. |
| shift | | Has methods to shift bits in a bit array. |
| logical | | Has methods to calculate "and" or "or" of two bit arrays. |
| range | | Has methods to obtain or modify bits within a range on a bit array. |
| XML | conv. | Convertible from/to XML strings. |
| booelan[] | | Convertible from/to a "boolean[]" object. |
| byte[] | | Convertible from/to a "byte[]" object. |
| file-io | | Can write to/read from a file. |
| copy | e. o. d. | Has methods (or constructors) to duplicate a bit array. |
| tostring | | Has a method of a "debug" print. |

*basic op. = basic operation, break-encap. = break encapsulation, conv. = conversion, e. o. d. = ease of development , impl. issue = implementation issue, rich op. = rich operation.*

The keywords extracted from source files were divided into equivalence classes: If keyword “a” appears in source files “f” and “g” only, and keyword “b” also appears in “f” and “g” only, then “a” and “b” are put into one equivalence class because they are equivalent in terms of their discriminative power. For each equivalence class of keywords, we need evaluate only one word from that class. In this case study, among the total 197 equivalence classes, 131 classes have only one word and the other 66 classes have an average 11.4 words, with the largest class having 236 words.

## 3.2 Overall evaluations

Table 2 shows queries with high accuracy (high precision and/or high recall) for each concept. We evaluated queries of one word, two words and three words. The queries shown in the table are queries of single word. If two or three word queries had higher precision and recall values than that of single-word queries for a particular concept, such queries are shown in the remark column in Table 2. Also, when the high discriminative queries in the second column are not intuitive ones, namely, those words seemed too difficult for a developer to guess from the concepts of the given

**Table 2.** High-accuracy queries for each concept

| **Requirement (concept)** | | **Max prec. * recall** | **Max prec.** | **Max recall** | **Remark** |
|---|---|---|---|---|---|
| | basic | { + } = 14/14 * 14/14 | ← | ← | **{ & } = 13/14 * 13/14** |
| scale | limited-size | { supplied } = 1/1 * 1/1 | ← | ← | **{ size } = 1/13 * 1/1** |
| scale | unlimited-size | **{ size } = 9/13 * 9/9** | { \|\| } = 5/5 | { size } = 9/9 | { limitations } = 2/4 * 2/9, { ++, size } = 9/12 * 9/9, { ++, copy, size } = 9/11 * 9*9 |
| scale | re-size | { initial } = 3/4 * 3/3 | { exceed } = 2/2 | { initial } = 3/3 | **{ size } = 2/13 * 2/3**, { initial, size } = 2/3 * 2/3 |
| impl. issue | non-pack | { name } = 2/2 * 2/2 | ← | ← | { size } = 2/13 * 2/2, { byte } = 0/9 * 0/2, { representing } = 2/3 * 2/2 |
| impl. issue | pack | { copy } = 12/13 * 12/12 | **{ ~ } = 10/10** | { copy } = 12/12 | { packed } = 0/4 * 0/2, { & } = 12/14 * 12/12. The equivalence class of "packed" includes "^". |
| impl. issue | mask | { prevent } = 2/2 * 2/2 | ← | ← | **{ mask } = 2/8 * 2/2**, { fast, mask } = 2/3 * 2/2 |
| impl. issue | break-encap. | **{ mutable } = 1/1 * 1/1** | ← | ← | The equivalence class of "mutable" includes "corruption", "performance", and "sanity". |
| rich op. | search | { serialized } = 3/3 * 3/3 | ← | ← | { first } = 3/5 * 3/3, { pos } = 3/5 * 3/3, **{ position } = 2/6* 2/3**, { <=, pos } = 3/3 * 3/3 |
| rich op. | merge | **{ merge } = 2/2 * 2/2** | ← | ← | |
| rich op. | value | { code } = 5/5 * 5/5 | ← | ← | **{ equals } = 5/7 * 5/5**, { !, * } = 5/5 * 5/5 |
| rich op. | shift | { <<= } = 2/2 * 2/2 | ← | ← | **{ shift } = 1/3 * 1/2**, { << } = 1/11 * 1/2, { >> } = 1/5 * 1/2, { >>> } = 1/5 * 1/2, { >>= } = { >>>= } = 1/1 * 1/2 |
| rich op. | logical | { ^= } = 3/3 * 3/3 | ← | ← | { and } = 3/13 * 3/3, { or } = 2/5 * 2/3, { nor } = 2/2 * 2/3, **{ and, or } = 2/5 * 2/3** |
| rich op. | range | { word } = 1/1 * 1/1 | ← | ← | **{ range } = 0/3 * 0/1**, { bounds } = 1/7 * 1/1 |
| conv. | XML | { replace } = 1/1 * 1/1 | ← | ← | **{ xml } = 1/4 * 1/1**, { string, xml } = 1/1 * 1/1 |
| conv. | booelan[] | { booleans } = 3/3 * 3/3 | ← | ← | **{ boolean } = 3/7 * 3/3** |
| conv. | byte[] | { gets } = 4/5 * 4/4 | { reserved } = 3/3 | { gets } = 4/4 | **{ byte } = 4/9 * 4/4** |
| conv. | file-io | { input } = 6/6 * 6/6 | ← | ← | **{ file } = 4/7 * 4/6**, { file, input } = 4/4 * 4/6 |
| e. o. d. | copy | { >= } = 8/10 * 8/8 | { >> } = 5/5 | { >= } = 8/8 | { copy } = 8/13 * 8/8, **{ clone } = 6/6 * 6/8**, { !=, >= } = 8/9 * 8/8, No three-word queryies outperformed. |
| e. o. d. | tostring | { string } = 11/12 * 11/11 | { / } = 9/9 | { string } = 11/11 | **{ to, string } = 11/11 * 11/11** |

*The "{...}" are queries. The values at right side of "=" are precisions and recalls. Each of these values is denoted by a fraction, whose denominator and numerator are counts of source files, without canceling down (reduction). A bold-font query is the query that looks the most intuitive one for the given requirement. A left arrow "←" means the query in the cell is the same to one in the left cell.*

requirement, the more intuitive queries were shown in the remark.

Recall values in Table 2 are relatively high, and this is not surprising for this study because each concept and queries are both extracted from the source files. In other words, it is guaranteed that some source files contain words of that concept. If the set of concepts were prepared without reading the source files, the recall values would be smaller.

Precision values varied among categories of concepts. From Table 2, we can find that the category of the ease of development have both intuitive and high-precision queries. The categories of implementation issue and rich operation, some concepts have both intuitive and high-accuracy queries while some don't. For the categories of conversion and scale, no intuitive and high-precision queries were found. For the category of basic operation, because 14 of total 15 source files have the concept, practically there is no need to query about this category.

### 3.3 Conclusions

The findings of the case study can be summarized as follows. Intuitive and high-precision queries are possible when (i) the name of the method that implement a concept is easy to guess from the coding convention of Java, such as copy → clone(), value → equals(), and tostring → toString(), or (ii) some unique words (operators) are required to implement the concept, such as, pack → ~ and shift → <<=. On the other hand, if the concepts are implemented without unique words, such as scale and conversion, it is difficult to find intuitive and high-precision queries.

**Acknowledgements.** I am deeply grateful for Dr. Yunwen Ye for his comments, advice, and proofreading. This research is supported by JSPS Grant-in-Aid for Challenging Exploratory Research (No. 21650008).

## References

1. O. Augusto, L. Lemos, S. Bajracharya, J. Ossher, "CodeGenie: a Tool for Test-Driven Source Code Search", Proc. ASE'07 pp. 525-526 (2007).
2. S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search", Proc. OOPSLA'06: pp. 681-682 (2006).
3. K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, S. Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search", Proc. ICSE'03, pp14-24, (2003).
4. D. Mandelin, L. Xu, R. Bodik, D. Kimelman, J. Mining, "Helping to Navigate the API Jungle", Proc. PLDI'05, pp.48-61 (2005).
5. Steven P. Reiss, "Semantics-Based Code Search", Proc. ICSE'09, pp. 243-253 (2009).
6. Y. Ye, G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information", Proc. ICSE'02, pp. 513-523 (2002).

The 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD2009), Tokyo, Japan, Nov. 2009.

# DesignMinders: A Design Knowledge Collaboration Approach

Gerald Bortis and André van der Hoek

University of California, Irvine
Department of Informatics
Irvine, CA 92697-3440
{gbortis, andre}@ics.uci.edu

**Abstract.** Software developers face numerous challenges in capturing and reusing knowledge during informal design sessions. While the knowledge that is brought to bear and generated during these sessions provides valuable insights that can ideally be reused, informal software design is a time when developers are gathered at the whiteboard working to solve problem and during which the developer must remain "in the moment" while engaged in discussion and sketching. In this paper we describe DesignMinders, a tool currently being developed to address these issues by augmenting an electronic whiteboard with the ability to capture, refine and explore design knowledge in the form of notecards. This paper documents our progress and describes several key challenges that we face.

**Keywords:** Software design, knowledge reuse, whiteboards, notecards

## 1. Introduction

Consider a team of software developers starting a new project. They hold a meeting to discuss the design and to look for some inspiration from solutions applied in other projects. They are not looking for a specific approach or technique; they just want to get a general idea of what others have done or problems they have encountered when working on similar projects. What if they had a way to easily explore this information?

Consider another group of developers gathered at a whiteboard in a meeting room working on a design problem. As they work through a potential solution, one of the developers recalls a certain constraint that had to be worked through the last time a similar problem was encountered. None of the developers can recall it, and the meeting stops. What if they were using a system that could automatically provide them with this information?

Finally, consider yet another group of developers, in the midst of session, when they realize that they are making some crucial decisions about the design that they would like to keep for later. What if they could easily capture these decisions, and have them presented when needed during subsequent sessions?

This is the space on which our research is focused: supporting software designers in capturing and using design knowledge in a lightweight way when they are designing at the whiteboard. This paper presents a vision of how this might be done, documents our progress towards it, and discusses several key challenges that we face.

## 2. Background

The above scenarios highlight obstacles that are commonly encountered by developers during informal software design. This is a time when developers are gathered at the whiteboard to better understand and work through a design problem. It is an activity that spans the development lifecycle and during which developers bring to bear knowledge from past projects or personal experiences that influence the decisions that are made [1-3]. The knowledge involved is at times explicit, easily conveyed, and about general software development practices, solution patterns, or a particular application domain. It can also be tacit knowledge that is specific to the organization or project [4]. As this knowledge is applied to the design, new knowledge is generated in the form of discussions and artifacts on the whiteboard that can provide valuable insights into how the system was designed; insights that in the ideal world are easily recalled during subsequent design sessions.

Unfortunately, as in the above scenarios, most of this knowledge “vaporizes” and fails to be reused after these design sessions, since it is difficult to capture and represent using existing approaches [5]. Informal design is a time when developers are engaged in ad hoc sketching and discussion, and are unlikely to follow a formal or prescribed process. It is crucial that developers be able to capture knowledge in a lightweight and informal way using a representation that allows for the knowledge to be refined at a later time. The developers also must be able to explore the collected knowledge and have it presented in a relevant way.

Existing techniques for capturing and reusing this design knowledge either fail to address these challenges altogether, or are focused on capturing knowledge during the more formal specification or implementation phases of the development process, when the informal design knowledge has already been lost. For example, knowledge management systems that allow developers to populate and search a knowledge base containing formal design documents fail to address the need for concise and relevant representations of knowledge during informal software design sessions when the design problem is still being understood. On the other hand, design rationale techniques attempt to capture the decisions that were made during a design session using an argumentation schema that is incompatible with the activities and processes that developers engage in while at the whiteboard. Such techniques are difficult to apply during informal software design when developers are engaged both with the whiteboard and with each other in discussion, and are unlikely to interrupt an opportunistic exploration of the design problem to populate the knowledge base with recently acquired knowledge about the design problem, or to formalize their partial solutions into an argumentation schema [6].

## 3. DesignMinders

To address these challenges, we present DesignMinders, a software design knowledge reuse tool to support developers in: capturing design knowledge in an easy way, organizing the collected knowledge as a set of notecards, and making this collection available for exploration and search during design sessions. We began by extending the electronic whiteboard and sketching tool Calico [7] that provides developers with distinct advantages over traditional whiteboards, the most important of which is the ability to directly manipulate sketches that are made on the whiteboard by selecting and moving them around. This allows for anything drawn or written on the whiteboard to become elements of design knowledge that can be built upon. As a result, we envision DesignMinders being used alongside Calico by developers during design sessions. To better understand how DesignMinders accomplishes this, imagine the following scenario.

A team of developers gathers to discuss the design of a new clinical application for viewing and managing patient account information. Before they begin sketching concepts for the user interface, they turn to the DesignMinders noteboard running alongside the electronic whiteboard and begin exploring the available notecards. They search for all cards tagged with the "ui" keyword, and are presented with a list of notecards. They quickly browse through the names of the notecards, reading the descriptions and glancing at some of the associated diagrams, and quickly assess if a notecard is relevant to their application. They drag-and-drop several relevant notecards from the search list to the noteboard and create a stack.

One of the selected notecards is named "Multiple Monitors" and describes how a user interface can be designed to take advantage of multiple displays by allowing certain elements to be expanded outside of the primary window. Similar to a design pattern, the notecard's name provides a brief description of the situation in which the knowledge can be applied. Brief, descriptive names allow for notecards to be easily identified when browsing a large collection. The body of the notecard contains the details of the design knowledge that is being captured, like a concise description of the commonly encountered situation and a suggestion for a possible solution. Realizing that the approach suggested by the notecard is relevant to their application since the primary window could potentially be cluttered with other information, they incorporate it into their discussion.

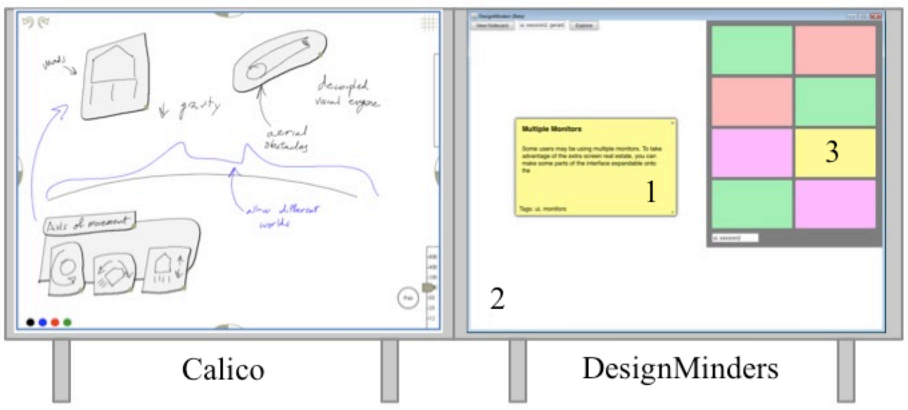
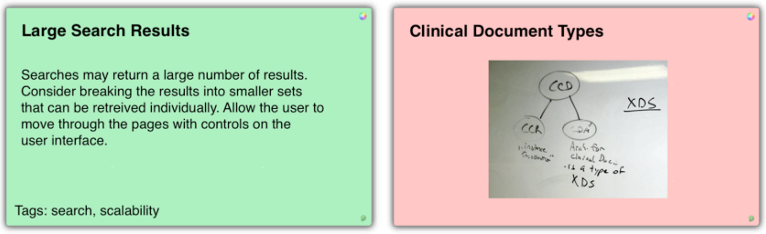


**Fig. 1** DesignMinders running alongside Calico on a pair of electronic whiteboards showing (1) a notecard, (2) the noteboard, and (3) the search list.

The developers then proceed to use Calico to draw sketches of what the user interface will look like, numerous class diagrams describing the model that will be used to represent the account information, and a list of tasks that will need to be done to complete this part of the project. They decide on a user interface that provides search functionality that populates a list of accounts. As they work through the design, DesignMinders is running on a secondary display alongside Calico and is monitoring their canvas actions for keywords that can be used to bring up notecards that are relevant to the design problem. It notices that the word "search" was written and automatically brings up any notecards that are tagged with the keyword. As the notecards are displayed on the noteboard, the title "Large Search Results" catches the attention of one of the developers. He selects the notecard to view more details and sees a description of a problem commonly encountered when displaying search results. The details of the notecard describes an approach to breaking the results into smaller sets that can be retrieved one at a time, a technique which the creator of the notecard had employed in a previous project. The reverse side of the notecard contains a class diagram created in a previous design session that roughly describes the implementation. Realizing that this problem most certainly does apply to their application, the developer brings up the notecard and the team re-evaluates their existing design to incorporate the pattern and address the issue.


Large Search Results

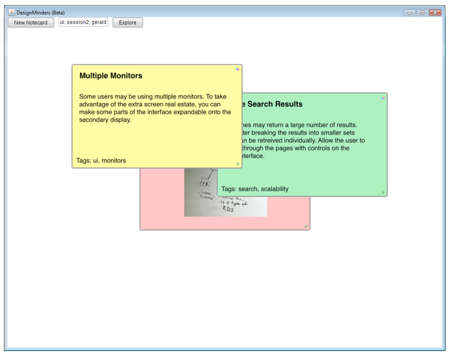
Searches may return a large number of results. Consider breaking the results into smaller sets that can be retreived individually. Allow the user to move through the pages with controls on the user interface.

Tags: search, scalability

Clinical Document Types

**Fig. 2** Notecards showing name, details, tags, and reverse side with image captured from whiteboard.

As the session progresses, the developers come up with numerous conceptual sketches of the user interface, brief notes on how patient account information will be retrieved, and some network diagrams describing the eventual deployment of the application. Realizing that these sketches and notes could be useful for future design sessions, they lasso several areas of the Calico canvas and select the "Create Notecard" function, which result in several new notecards in the DesignMinders interface. The developers decide that the user interface notecards should be colored green, the patient account notes yellow, and the deployment diagrams red. The notecards are automatically tagged with the day's date and the internal name of the project. With the cards now on the noteboard, they quickly go through each one and give it a name and provide additional details or tags. The following week, the developers return to the conference room to further work out details of the design. They use the previously created notecards as a starting point to continue their discussion.



Multiple Monitors

Some users may be using multiple monitors. To take advantage of the extra screen real estate, you can make some parts of the interface expandable onto the secondary display.

Tags: ui, monitors

e Search Results

hes may return a large number of results.
der breaking the results into smaller sets
n be retreived individually. Allow the user to
through the pages with controls on the
terface.

Tags: search, scalability

**Fig. 3** Noteboard with several notecards.

## 4. Discussion

DesignMinders faces the knowledge reuse challenges that exist during informal software design by presenting developers with a lightweight means of capturing, representing, and applying design knowledge. This is partially accomplished by representing bits of knowledge using notecards that mimic physical index cards. Physical cards lend themselves to easy browsing, manipulating, annotation, and ad hoc organization through stacks. Our goal was to reproduce this interaction in an electronic form to take advantage of the indexing and searching that can be performed. The ease with which a notecard can be created "in the moment" and the fact that none of the elements of a notecard (name, details, and tags) are required significantly lowers the barrier to capturing important knowledge during design sessions. The conciseness of the notecards also lends themselves to quick browsing to determine relevancy to a particular design. Our goal of a lightweight knowledge retrieval tool is also accomplished through the use of a search interface that allows for quick browsing of collected notecards and a reduction of the search space through the use of tags and colors to indicate significance or categories. The noteboard also allows for the ad hoc creation of localized groups by stacking relevant cards, much like one would create a stack of index cards.

## 5. Challenges

We see much promise in DesignMinders as a design knowledge reuse tool that can aid in collaboration during software design sessions and answer the "what if?" questions. Our lightweight and informal approach to knowledge reuse presents several challenges for future work.

The first challenge is to continue to lower the barrier to quickly and easily capturing knowledge by improving the integration between the ongoing design and the knowledge representation. This includes the current ability to manually select elements of the Calico canvas and create new notecards on the noteboard, as well as to have notecards be created automatically based on groupings or certain criteria such as the amount of time spent on a specific canvas. Also, we would like to increase the amount of information that is automatically collected from Calico by taking advantage of contextual elements, like UML diagrams and lists, to automatically populate detail and tag fields on new notecards.

The second challenge is to allow developers to easily annotate the design with reusable knowledge. Notecards are currently retrieved by browsing a list that allows for filtering by any of the fields on the notecard as well as the color. Relevant cards can be dragged onto the noteboard to create stacks of cards. We are exploring the ability to drag-and-drop cards from the DesignMinders interface directly into the Calico canvas, like attaching a sticky-note to a whiteboard. This would allow for the notecards to essentially become part of the design, and even become rationale for certain design decisions that are made.

The third challenge is to improve the relevance of the knowledge that is presented to the developer during a design session. This includes adding the ability to recognize words and phrases written on the canvas and present the designer with relevant cards based on keywords. This further reduces the amount of work that the developer must do to locate prior knowledge. There is also the potential to find relevant cards based on similar diagrams. For example, if a UML diagram is drawn on the Calico canvas while in UML mode, notecards with similar diagrams could be retrieved and displayed alongside, or even directly incorporated into the canvas as reusable palette elements.

## 6. Conclusion

A prototype version of DesignMinders is currently being developed. While the basic functionality that we have described provides a novel approach for capturing and presenting knowledge during informal software design sessions, DesignMinders can be improved by addressing the challenges presented. We began by posing several scenarios in which developers were hindered in their ability to capture and recall previously gained insights during a software design session. We see DesignMinders as a way to remove these barriers to knowledge reuse while addressing the needs of the developers in capturing and presenting knowledge during this distinct activity.

**Acknowledgments.** Effort partially funded by the National Science Foundation under grant number 0920777.

## References

[1] G. Fischer, "Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments," *Automated Software Engineering,* vol. 5, pp. 447-464, 1998.

[2] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: how and why software developers use drawings," *Proceedings of the SIGCHI conference on Human factors in computing systems,* 2007.

[3] A. Murray and T. C. Lethbridge, "On generating cognitive patterns of software comprehension," *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research,* 2005.

[4] I. Frank M. Shipman and C. C. Marshall, "Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems," *Computer Supported Cooperative Work,* vol. 8, pp. 333-352, 1999.

[5] R. Farenhorst, "Tailoring knowledge sharing to the architecting process," *SIGSOFT Software Engineering Notes,* vol. 31, p. 3, 2006.

[6] A. Aurum and M. Handzic, *Managing Software Engineering Knowledge*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[7] N. Mangano, A. Baker, and A. v. d. Hoek, "Calico: a prototype sketching tool for modeling in early design," *Proceedings of the 2008 international workshop on Models in software engineering,* 2008.

# On the Use of Emerging Design as a Basis for Knowledge Collaboration

Tiago Proenca, Nilmax Teones Moura, and André van der Hoek

University of California, Irvine, Department of Informatics, Irvine CA 92697, USA
{tproenca, nmoura, andre}@ics.uci.edu

**Abstract.** Abstractions in software engineering have been used for guidance and understanding of software systems. Design in particular is a key abstraction in this regard. However, design is often a static representation that does not evolve with the code and therefore cannot help developers in collaborating after it becomes out-of-date. Our research group is exploring the use of Emerging Design, a dynamic abstraction, as the basis for knowledge collaboration through its implementation in a coordination portal called Lighthouse. This paper presents the state of the art of Lighthouse and discusses three knowledge collaboration problems that we are currently addressing.

## 1 Introduction

Collaboration is related to mutual sharing of knowledge [1] and has become an essential part of software development and indeed an important research field in software engineering. Today, most knowledge sharing is either informal or decoupled from the actual artifacts to which it pertains. For instance, in the Knowledge Depot [2], an email-based group memory tool, knowledge is stored in a separate repository that must be queried to find a particular piece of information. This not only creates a hurdle to accessing knowledge, but also leads to the update problem, i.e., in the presence of changes, one has to update two places: the artifacts themselves and the Knowledge Depot.

Our research group is exploring a different kind of solution, one where the knowledge is essentially attached to an abstraction that we are creating as part of a collaboration infrastructure. This abstraction is called Emerging Design [3] and is defined as the design representation of source code as it changes over time. With each code change, the Emerging Design is updated accordingly. Emerging Design satisfies the traditional roles of abstraction (guidance and understanding) and includes support for new roles such as coordination, project management, and detection of design decay.

While the original focus of our use of Emerging Design was on detecting conflicts in code changes, it seems a particularly promising abstraction to address a broader class of collaboration issues. In this paper, we talk about three such collaboration problems and how we believe Emerging Design serves as good basis for exploring them.

The remainder of this paper is organized as follows. In Section 2, we review Emerging Design and its implementation in Lighthouse. Section 3 presents three knowledge problems in collaboration and how we believe they can be addressed by building upon Emerging Design. In Section 4, we summarize our idea and discuss some challenges and future directions to improve our work.

## 2 Emerging Design

Since a design document illustrates the interactions among modules, it can help developers to gain an understanding of the high-level structure of the system and its interactions [3, 4]. However, design is often a static representation that does not evolve (automatically) with the code. Therefore, as it becomes out-of-date, it loses value for developers who need to collaborate.

Our research group is exploring the use of Emerging Design as the basis for collaboration. Emerging design is defined as the design representation of source code *as it changes over time.* It is a live document that stays up-to-date with all changes made to the system. It is annotated with information about the changes made, helping developers to be aware about how the code structure evolve, and with whom they may need to coordinate their actions in order to reduce and prevent conflicts.

We implemented this idea as an Eclipse plug-in called Lighthouse [5]. Currently, Lighthouse has only the Emerging Design view that shows a UML-like class diagram which is built dynamically as developers implement or make changes in the code. One particular characteristic of Lighthouse is that it does not require check-in of the changes made. Instead, it tracks workspaces, since the goal is help people collaborate and coordinate before sending the changes to the source code repository, so merge conflicts are avoided.

Figure 1 shows the Emerging Design basic representation. It shows the primary elements found in UML class diagrams, such as classes, fields, and relationships, as annotated with additional information. In particular, Lighthouse shows information about the evolution of the code. The *plus* symbol represents an addition of a class/method/field, *minus* represents a removal and *triangle* represents a change. For instance, in the `ATM` class, the field *value* was added by Max. Another example is the field *balance_inquiry.* We can see that Theo and Bob changed that field, and finally Anna removed it. Notice that the evolution or history of changes is presented in a top-down manner, time-ordered with the most recent changes in the bottom.

The use of Lighthouse in a large software product naturally introduces scalability issues with respect to the visualization. In order to address this problem, we are developing a variety of filters. With these filters, the information shown can be reduced by focusing on particular packages, developers, or some combination of them. Figure 2 shows an Emerging Design representation with several classes, where each class has numerous events representing the activities of four developers (Max, Bob, Ana, Jim), who are all coding a particular part of the project. The following picture (Figure 3) illustrates how Lighthouse allows users

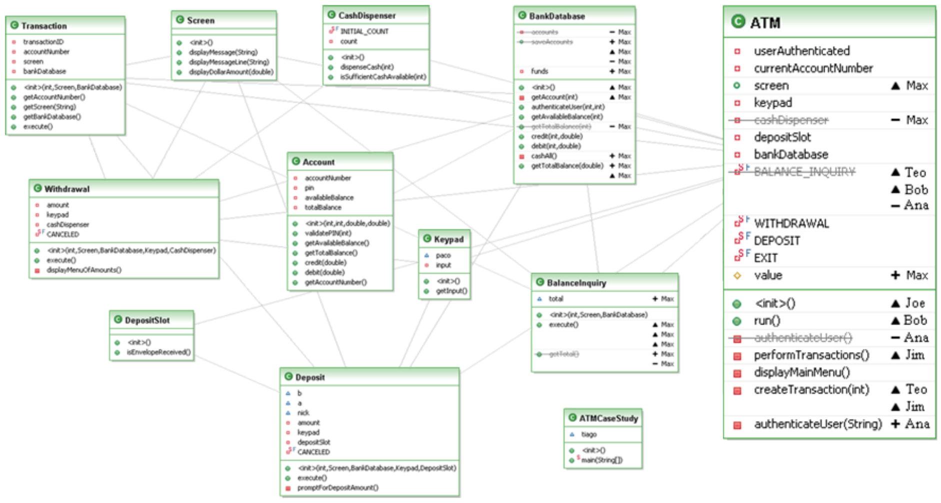


**Fig. 1.** Emerging Design Basic Representation. ATM class blows up to show detail.

to turn on the filter by developer. In this specific example, the user has chosen to show Bob's code changes. Other filters exist as well. As a result, crowded visualizations that clearly indicate a problem can be examined for what that problem exactly is.

To date, Lighthouse is a collaboration portal focused on detecting conflicts. It uses the Emerging Design concept to show who is making the changes where, and by looking at that, enable developers to find where their changes may be conflicting with somebody else's. In this paper, we take this work a step further. We outline how we believe the concept of Emerging Design is not only useful for detecting conflicts, but also as a basis for knowledge collaboration. In the next section we talk about three particular knowledge collaboration problems and how Emerging Design can be used as a basis for exploring them.

## 3 Three Knowledge Problems

Knowledge collaboration manifests itself in many different forms and may revolve around many issues. In this section, we discuss the following three problems: (1) How to support developers in determining where the implementation is deviating from the original design; (2) How to support finding the right expert related to a given design; and (3) How to support identification of those parts of the program with less than ideal quality.

### 3.1 Design Decay

It is well known that software changes, and that such changes involve modifications to the original design that may lead to design decay [6]. Prior to the

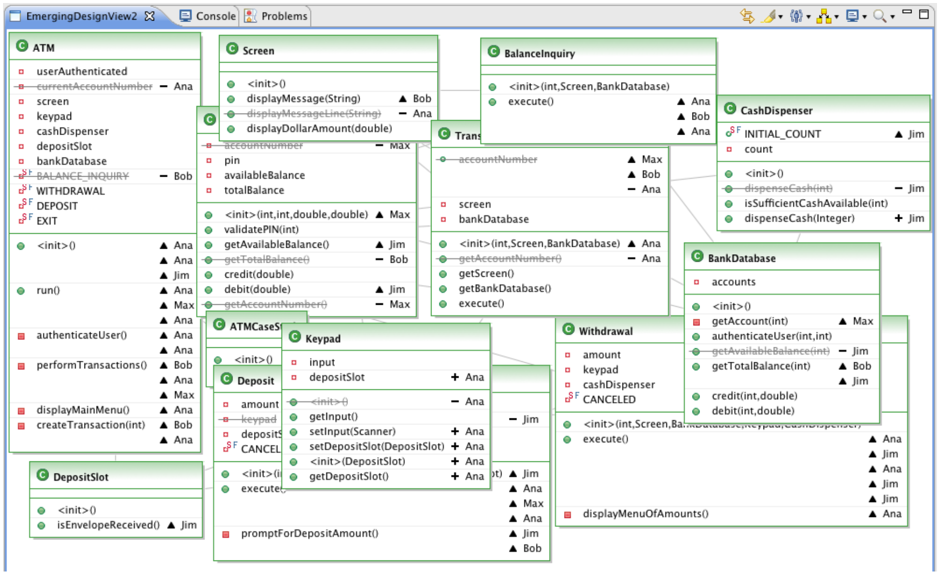


**Fig. 2.** Emerging Design Basic Representation with four developers (Max, Bob, Ana, Jim).

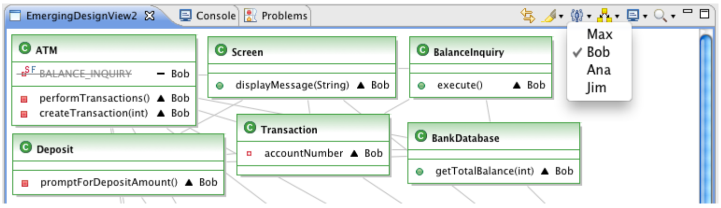

EmergingDesignView2
Console
Problems
ATM
~~BALANCE_INQUIRY~~ – Bob
performTransactions() ▲ Bob
createTransaction(int) ▲ Bob
Screen
displayMessage(String) ▲ Bob
BalanceInquiry
execute() ▲ Bob
Max
✓ Bob
Ana
Jim
Transaction
accountNumber ▲ Bob
BankDatabase
getTotalBalance(int) ▲ Bob
Deposit
promptForDepositAmount() ▲ Bob

**Fig. 3.** Emerging Design Basic Representation with filtering. Just modifications in Bob's workspace are shown.

implementation phase, some conceptual design diagrams are usually constructed to guide developers and help them understand the project's high-level picture. The reasons *why* a particular design decays generally are not available, and therefore could be said to represent a knowledge collaboration problem. In future, other developers must understand why a certain piece of code is like it is, and much rationale resides behind the code changes from the original design to the current state.

We can address this issue by marking the Emerging Design, so it shows deviation, and providing facilities for developers to provide contextual information pertaining to the changes they make. Imagine a developer restructuring a certain piece of code in a certain way that is counter-intuitive. By leaving a note, directly visible on the diagram (Figure 4) they now can motivate their change. Other developers can respond either in the affirmative or by expressing concerns

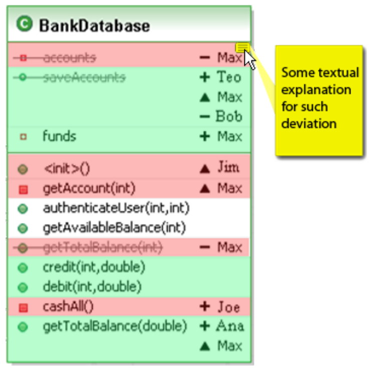

BankDatabase
~~accounts~~ − Max
~~saveAccounts~~ + Teo ▲ Max − Bob
funds + Max
<init>() ▲ Jim
getAccount(int) ▲ Max
authenticateUser(int,int)
getAvailableBalance(int)
~~getTotalBalance(int)~~ − Max
credit(int,double)
debit(int,double)
cashAll() + Joe
getTotalBalance(double) + Ana ▲ Max
Some textual explanation for such deviation

**Fig. 4.** Design Decay Representation.

and such. A discussion can ensue, for which it is crucial to note that the discussion takes place directly in the context of the artifacts and as the changes are happening. Design decay can be avoided this way, and design evolution becomes under joint ownership of the developers.

In Figure 4, the green overlays are used for elements that are present in both conceptual and emerging design, i.e., the ones that were implemented according to the original design. Red overlays are used on items that are in the emerging but not in the conceptual design, meaning that the implementation diverges from the original design. Elements left in white are the ones that are in the conceptual but not in the emerging design. These elements have not been implemented yet.

The Emerging Design provides a natural basis for addressing design decay because it already tracks design evolution. By now using this basis with simple but powerful extensions, the Emerging Design provides instant knowledge collaboration, both implicitly because it makes visible the design as it evolves and explicitly because its evolution can be gauged, questioned, discussed, and resolved as needed.

We also note that this can take place both among individual developers at the level of individual or small sets of changes, and by team leaders and architects based upon views of the code as a whole.

## 3.2 Expertise

The time taken to find an expert is one of the major reasons that co-located work tends to take less time than similar development work split across sites [9]. Quickly finding the right expert related to a given design and/or implementation issue is critical to the success of any software development project. There is a clear knowledge collaboration problem when one needs to understand how some class/method works, why it is as it is, and how it may need to evolve. Often, an expert can provide useful assistance in this regard.

We again explore how the basis of Emerging Design can be leveraged to address this problem. Particularly, we envision exploring the use of a visualization

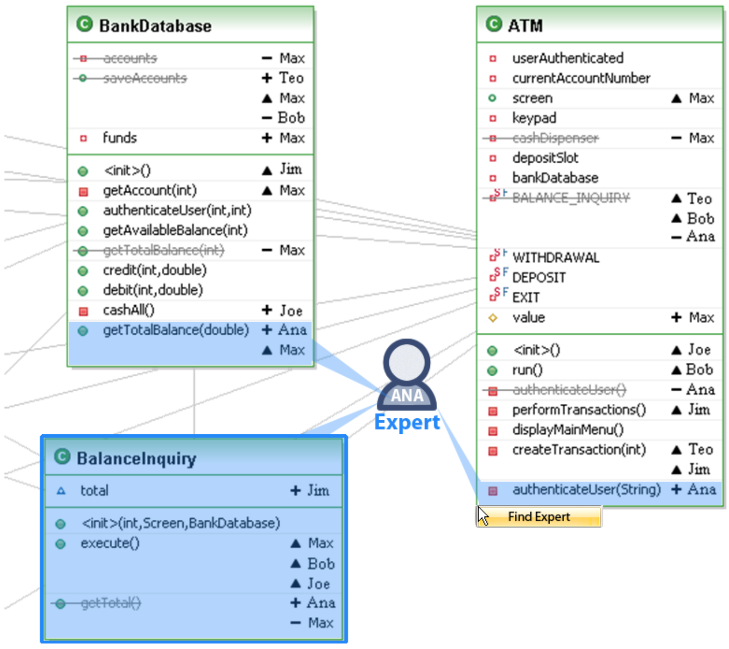


**Fig. 5.** Expertise Representation.

to allow users to browse through the Lighthouse diagram in order to find the proper expert. Since Lighthouse already provides the basis for who made which changes, now we can actually build various overlays that make it possible, for instance, to click on one of the authors on a particular method and have the other pieces that they changed highlighted.

In another form, we note that it is often difficult to find someone with broader knowledge pertaining to multiple artifacts and methods. We plan to develop a feature that allows the user to select a group of methods and classes in order to find the expert related to those couple of artifacts, as shown in Figure 5.

The advantage here is that, while most expertise systems are limited to work at the level of artifacts, our approach can provide more fine-grained as well as a broader range of answers.

## 3.3 Code Quality

Software quality metrics can drive software process improvements [7]. Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs [8]. Some information that could be useful in this regard is the overall quality of each class, which if available would enable the identification of the most problematic or complex parts of a project. This kind of information is not usually accessible, representing the third knowledge collaboration problem that we address in this paper.

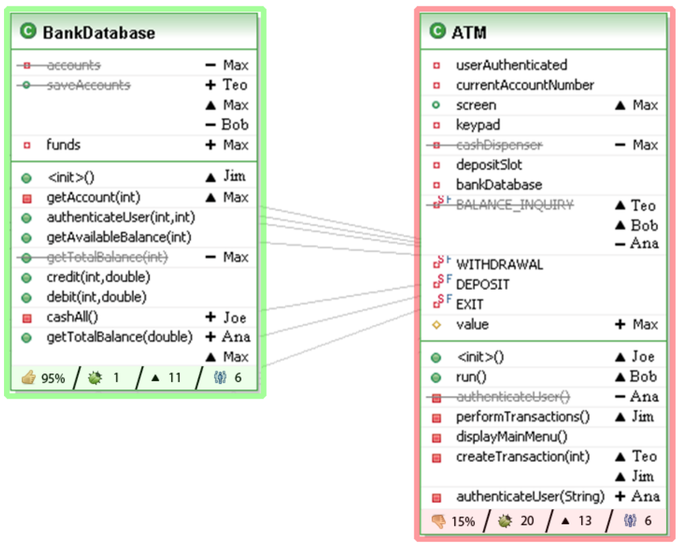


**Fig. 6.** Code Quality Representation

The use of Emerging Design in this situation would help developers and managers to quickly spot code that is growing without proper quality. Then we envision a software quality visualization that will show individual factors, such as *number of developers*, *number of recent bugs*, *how well the class/method was tested*, and *number of changes/code volatility* at the bottom of each class. We also take these individual factors in consideration to provide an overall quality measure, and we represent this high level awareness information by using color border scale, in which green means good quality and red means bad quality (Figure 6).

We increased the capability of Emerging Design to deal with software quality considerations. This approach is good for understanding which classes/methods are producing higher quality code and also a project's progress in the quality perspective over time. In this way, managers would be able to identify areas that need attention, and for example tell what parts of the project are in need of more tests and what parts have enough coverage already.

## 4 Related Work

Several tools have been created to help people collaborate and to enhance individuals' awareness. The War Room Command Console [9] shows in a public display the current state of a system across workspaces in real-time. The visualization shows the ongoing changes made by developers in thumbprints, a graphical representation of the source code, displayed in a topographic layout. This work, like Lighthouse, uses a program-centered approach to show how changes made by developers are related with the artifacts and how the system is evolving. Its display, however, is in a central location and not on a per-developer basis. Furthermore, the information that it shown is compacted, and does not allow easy access to details.

Palantír [10] provides real-time awareness of changes made by developers and estimates the impact of how severe these changes are. Palantír, like Lighthouse, does not require developers to check-in the changes made and presents a view with information of all developers' workspaces. However, Palantír differs from Lighthouse since it uses a low-level abstraction that focuses on files, while Lighthouse uses the concept of Emerging Design.

FastDash [11] and CollabVS [12] both use a collaboration-centered approach to display the artifacts' interaction among developers. Unlike Lighthouse, this approach uses real-time awareness of developers' activities instead of focusing on program artifacts. The visualization shows people and the activities they currently undertake, e.g., who has which file open or who is editing which file. This approach has the drawback of not providing a spatial awareness of artifacts and it does not provide an historical view of changes made.

## 5 Summary

In this paper we recapped Emerging Design and presented our vision of the potential role it can play in knowledge collaboration. We described Lighthouse briefly and addresses three knowledge collaboration problems: *design decay* by providing developers with the rationale resides behind the code changes from the original design to the current state; *expertise* by finding the proper expertise for a particular group of methods and/or classes; and *code quality* by providing developers the identification of parts of the program with less than ideal quality.

The benefit we can see is that the knowledge is directly anchored to the artifacts to which it pertains and is thereby easily accessible and intuitive since it fits with the task that a developer is currently working on. Presently, we are engaged in providing this support and we will perform various explorations and evaluations as we build our extensions to Lighthouse. A particular question is whether Emerging Design is useful to support other knowledge collaboration problems. Another question is how it can support multiple problems in parallel, as some of our solutions use similar techniques and thus cannot be used at the same time.

## 6 Acknowledgments

Effort partially funded by the National Science Foundation under grant number 0920777.

## References

1. Rus, I., Lindvall, M.: Knowledge management in software engineering. IEEE Software **19**(3) (2002) 26–38
2. Kantor, M., Zimmermann, B., Redmiles, D.: From group memory to project awareness through use of the knowledge depot. In: CSS '97: California Software Symposium. (1997)
3. Van der Westhuizen, C., Chen, P.H., van der Hoek, A.: Emerging design: New roles and uses for abstraction. In: ROA '06: Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering, New York, NY, USA, ACM (2006) 23–28
4. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Transaction on Software Engineering **12**(2) (1986) 251–257
5. da Silva, I.A., Chen, P.H., Van der Westhuizen, C., Ripley, R.M., van der Hoek, A.: Lighthouse: Coordination through emerging design. In: Eclipse '06: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange, New York, NY, USA, ACM (2006) 11–15
6. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering **27**(1) (2001) 1–12
7. Livingston, J., Prosise, K., Altizer, R.: Process improvement matrix: A tool for measuring progress toward better quality. In: Proceedings of 5th International Conference on Software Quality. (1995)
8. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: ICSE '76: Proceedings of the 2nd International Conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 592–605
9. O'Reilly, C., Bustard, D., Morrow, P.: The war room command console: shared visualizations for inclusive team coordination. In: SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, New York, NY, USA, ACM (2005) 57–65
10. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantír: raising awareness among configuration management workspaces. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 444–454
11. Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G.: Fastdash: a visual dashboard for fostering awareness in software teams. In: CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, ACM (2007) 1313–1322
12. Hegde, R., Dewan, P.: Connecting programming environments to support adhoc collaboration. In: Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on. (2008) 178–187

# A Proposal of TIE Model for Communication in Software Development process

Masakazu Kanbe[1,2], Shuichiro Yamamoto[1] and Toshizumi Ohta[2]

[1] Research Institute for System Science, NTT DATA CORPORATION, 3-3-9 Toyosu Koutoku Tokyo, Japan
[2] Graduate Department of Social Intelligence and Informatics, Graduate School of Information Systems, The University of Electro-Communications, 1-5-1 Choufugaoka, Choufushi, Tokyo Japan

kanbems@nttdata.co.jp, yamamotosui@nttdata.co.jp, ohta@is.uec.ac.jp

**Abstract.** Communication is more important in software development fields. We proposed the intermediary knowledge model to analyze the enterprise communication by extending traditional knowledge creation model. In this article, we propose TIE models based on intermediary knowledge. TIE model is the knowledge network model to explain the just in time documentation in the CMC tools like wiki. We analyzed the case of wiki based software development and showed the effectiveness and efficiency of the CMC tools in software development in certain conditions.

**Keywords:** Knowledge communication, Software development, Knowledge network

## 1 Introduction

Software developments become more complex and lots of people, which has various backgrounds participant in its processes. Various communications occurred in the field of the software development. The communication style of software developments contains regular meetings, ad hoc conversations in the local office, and acceptances of document by e-mail. Furthermore, CMC (Computer Mediated Communication) tools such as wiki, SNS, blogs and communication plug-ins of Integrated Development Environment support the developers' communication. In this article, we propose TIE model as knowledge network model for software development communication. TIE model is three layered network model. The three layers are tacit knowledge network, intermediary knowledge network and explicit knowledge network. We analyzed the case of wiki used software development process by TIE model. We also investigate the effectiveness and efficiency of wiki used software development process.

# 2 Related Works

In this chapter, we introduce the previous related works to explain our model.

## 2.1 Intermediary knowledge model

We proposed the intermediary knowledge model as knowledge sharing model in enterprises [1] [2]. Intermediary knowledge is the knowledge statement in which employees share the knowledge by the CMC tools. Intermediary knowledge model is one of the extended models of tacit and explicit knowledge concept [3].

The traditional knowledge creation model has tacit and explicit knowledge and four knowledge transformation modes; socialization, externalization, combination and internalization [4]. Intermediary knowledge model is proposed to solve problems and perform business tasks without the knowledge spirals of the organizational process. Employees can share the knowledge in intermediary knowledge statement by using CMC tools. In intermediary knowledge model, employees can exchange the knowledge that cannot be shared in tacit knowledge with less cost compared to explicit knowledge.

Fig. 1 shows the intermediary knowledge model. The dashed lined square in Fig. 1 indicates the traditional knowledge creation model. We add the intermediary knowledge to the traditional model. This model indicates that the employees can rapidly develop the knowledge in the CMC tools by using the intermediary knowledge transformation modes. The modes consist of publication, fragmentation, collaboration, resonant formation, and sophistication. Publication means to publish individual experience or ideas. Fragmentation means to import the parts of explicit knowledge. Collaboration means to react with employees' problems or opinions. Resonant formation means to accept and understand the others' opinions. Sophistication means to develop explicit knowledge from intermediary knowledge.

According to the traditional knowledge spiral condition, if employees intend to use the knowledge formally and inter-organizationally, each organization have to generate the explicit knowledge through the inner organizational knowledge spirals. Formally making explicit knowledge needs high cost and much labor through the inner organizational knowledge spirals. Intermediary knowledge transformation modes explain lower cost and labor in the knowledge exchange than the traditional knowledge transformation modes.

Also intermediary knowledge model explains the effectiveness of communication records. CMC tools create the more interaction points for employees than they do not use CMC tools. The employees have new communication in CMC tools. The communication in CMC tools are recorded as intermediary knowledge and employees reused the knowledge efficiently.

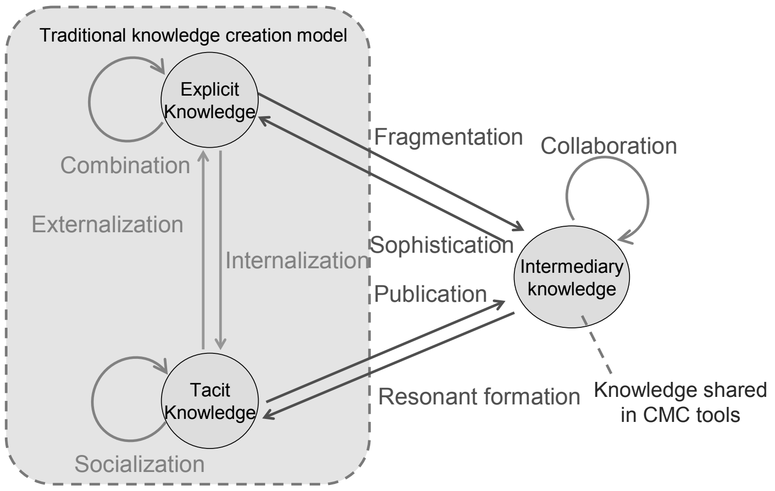


Fig. 1 Intermediary knowledge model

### 2.2 IBIS model

IBIS [5] and gIBIS [6] [7] are the traditional software engineering method. One of the purposes of these IBIS methods is to fully record and structuralize the discussion processes and progresses in software developments. Recording and structuring all the discussion processes and progresses, software developers could find the important information in developments. Although the records and structures of IBIS or gIBIS may be useful, the costs or labors are too large to make and reuse the full documented records.

### 2.3 Recent software engineering researches

Software engineering researches supports the software developers' communication. Software developments need the knowledge communication among software developers. Ko et al. [8] analyzed the software developers' activities and found that the developers used coworker as information source. This research indicates that the communication among the developers is very important in recent software developments.

Ye et al. [9] [10] helped the software developers to search the knowledge from the software libraries and members of software development team. They proposed the personalized search engine for API documents and communication channels for experts in software development team. Their researches implicate the way to make developers communicate each other for efficient software development.

Marczak et al. [11] indicated the importance of information brokers in requirement change management. As their research, the information brokers have important roles in the social network of software development team. The information brokers facilitate information flow to avoid misinterpretations of requirements. These researches indicate the importance of the developers' communication in software development.

## 3 TIE model

### 3.1 Overview of TIE model

We propose TIE model as CMC model for dynamic communication in software development process. TIE model has three layers consisted of Tacit Knowledge Network (TKN), Intermediary Knowledge Network (IKN) and Explicit Knowledge Network (EKN). Table 1 shows features of these three layers. Table 1 indicates the definitions of TIE model.

**Table 1.** Features of layers of TIE model

| Knowledge Network | Network node | Media | Documentation | Examples of products |
|---|---|---|---|---|
| Tacit Knowledge Network | Human | Face to Face, Telephone, Video conference | No documentation | Discussions, Meetings |
| Intermediary Knowledge Network | CMC content | CMC tools | Just in time documentation | CMC logs |
| Explicit Knowledge Network | Document | Document management services | Full documentation | Requirements, Specifications, Source codes, Manuals |

TKN has roles to exchange the tacit knowledge. The network node of TKN is human. TKN is related to organization structures, roles of members, processes of decision making and so on. TKN is occurred in face to face meeting, telephone or video conference communication. It seems that TKN brings down no documentation for the software development. We assume TKN does not create any formal documents. The products of TKN are discussions and meetings. TKN does not always create the tangible products to be observed.

IKN has roles to exchange the intermediary knowledge. The network node of IKN is CMC content. IKN is related to CMC network in the software development team. These CMC contents grow up in CMC tools such as Wiki, Blog, and SNS. IKN provides just in time documentation with the developers. If one needs to coordinate with others, one can use CMC tools to coordinate with others. And the coordination records are published for all the members of software developmental teams. These published coordination records are useful documents for software development. We call this process "Just in time documentation." Just in time documentation means that the necessary knowledge becomes documents when the developers communicate each other in CMC tools. The products of IKN are CMC logs.

EKN has roles to exchange the explicit knowledge. EKN is related to document network in the software development process. The network node of EKN is document. This document network grows up in document management services, which of functions are the document traceability, the historical management, the full text search and the file sharing. EKN provides full documentation with developers. The products of EKN are documents, such as requirements, specifications, source codes, manuals and guidelines.

The network edges of TKN mean human oral communication. The edges of IKN mean the concatenations of CMC contexts. The edges of EKN mean the interrelationship among documents. The edges between TKN and IKN mean the processes of intermediary knowledge provisions and acquisitions via CMC tools. The edges between IKN and EKN mean the processes of quotations and documentations of explicit knowledge via CMC tools.

### 3.2 TIE model for software development communication

TKN do not create any formal document. We call this TKN statement "no documentation." EKN aims to create the documents elaborately. We call this EKN statement "full documentation." Traditional software developments use the TKN and EKN as the knowledge process.

However, the knowledge processes in the traditional software developments has two problems. First problem is the loss of the important information of software developments. The knowledge processes in TKN are communication in discussions and meetings. Communication records of TKN are almost disappeared when the meetings or discussions ended. Communication contents of TKN are not always described in documents and merely shared with all the members.

Second problem is the difficulty to record the all the important information of software developments. If all the events in software developments were documented fully at right time, each member could understand requirements, specification, and source codes perfectly. However, it is difficult to achieve full documentation because its cost is very high and its range is very ambiguous.

Fig.2 shows TIE model we proposed. TIE model adds IKN to the traditional knowledge process in software developments. CMC tools support IKN. Balloons express the representative knowledge process of TIE model. The square balloon means the knowledge processes of traditional software development style. Round balloon means the knowledge processes of particular TIE model.

The knowledge processes in IKN are open and agile communication on CMC tools. The open CMC tools facilitate the communication of software development teams. IKN records the CMC logs. These CMC logs are not formal document, but very useful knowledge for software development. The development team members can read the knowledge processes each other in CMC tools. The knowledge processes in

TIE model have correspondence relation with the knowledge transfer modes in the intermediary knowledge model in Fig. 1. Tacit knowledge, intermediary knowledge and explicit knowledge are corresponded with TKN, IKN and EKN respectively.

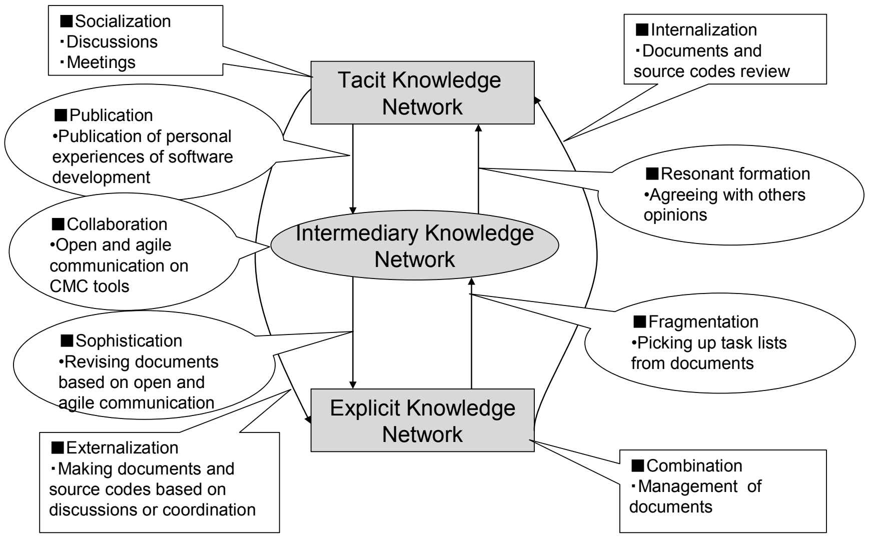

■Socialization
・Discussions
・Meetings
Tacit Knowledge Network
■Internalization
・Documents and source codes review
■Publication
•Publication of personal experiences of software development
■Resonant formation
•Agreeing with others opinions
■Collaboration
•Open and agile communication on CMC tools
Intermediary Knowledge Network
■Fragmentation
•Picking up task lists from documents
■Sophistication
•Revising documents based on open and agile communication
Explicit Knowledge Network
■Externalization
・Making documents and source codes based on discussions or coordination
■Combination
・Management of documents

Fig.2 TIE model and knowledge processes

## 4 Case Study of TIE Model for Software Development

To confirm the effectiveness and efficiency of TIE model, we analyzed the case of the wiki based software development. Wiki was used to facilitate the communication in software development case.

### 4.1 Aim of the case study

The case study aims to verify assumptions below;

- A1: CMC tools can record the important knowledge for the software development.
- A2: CMC tools can facilitate to share knowledge which did not efficiently share in traditional software development style.
- A3: CMC tool can provide the appropriate communication occasions.

These assumptions are set to confirm the effectiveness and efficiency of CMC tools in software development. We define that to verify these assumptions are to verify the effectiveness and efficiency of TIE model in software development communication because CMC tools support IKN.

### 4.2 Overview of case

We selected the wiki based software development case. The number of software developers in this case was nine. Nine members belonged to two other companies. They cooperated to develop the software system development with unfamiliar devices. This software development had the processes; document production, program coding, and program test. These two companies office were in different location with no time zone difference. Although the members had the face to face meetings at once a week regularly, some communication mistakes occurred and caused negative effects for the software development. To deal with the communication mistakes, this software development team determined to use wiki to compliment with team communication. Appendix shows the outlines of the wiki communication. This wiki had 13 pages. There were 21 items in 13 wiki pages.

We observed the knowledge processes of TIE model, 18 publications, 5 fragmentations and 2 collaborations in wiki. An example of the publication was W3 in Appendix. The case of W3 was that a member imagined the tasks to be done for this process and published them. An example of fragmentation was W12 in Appendix. The case of W12 was that a member extracted documents lists from the development conference regulations in this organization. An example of collaboration was W8 in Appendix. The case of W8 was that members communicated about the test policy in wiki. We did not observe the knowledge process of resonant formation and sophistication. These two knowledge processes might occur outside wiki in this case.

### 4.3 Verifying the assumptions

To verify the assumptions, we picked up the evidences from the CMC in the wiki.

- Verifying A1: CMC tools can record the important information for the software development.

This assumption is related to “recording.” The wiki recorded the important information for the software development. Nine members used the wiki and recorded the 21 important knowledge items for the software development. For example, there were design documents lists in W1 and W2 of Appendix. These lists were fragmented intermediary knowledge from the explicit knowledge such as document inventories. A member considered that it is necessary for the software development team to share the documents lists. This member extracted the document names from the document inventories.

This member did not use e-mail to share the document lists, but used wiki. Other members added the progress information with the document items in wiki. To add the progress information is publication of intermediary knowledge. All the member shared the dairy progress information of each document by wiki. The knowledge in the wiki is open to all the members and shared one target. If they shared this progress information by e-mail, they would not continue to refine the progress information because e-mail has the feature of cross in the post.

- Verifying A2: CMC tools can facilitate to share information which did not efficiently share in traditional software development style.

This assumption is related to “effective knowledge sharing.” The wiki facilitated to share knowledge which did not efficiently share in traditional software development style. “Development know-how” in W15 of Appendix is one of the evidences for A2.

This development know-how was published by a member who had a similar development experiences. This member wrote politely the knowledge to treat with specific devices of such as tips of particular sensors or actuators control. This knowledge was based on the member’s own experiences and not formalized yet. In traditional software development, such know-how may be orally transferred among developers who joined the oral communication in TKN. In this case, the knowledge sharing in the wiki might avoid the rediscovery of this the knowledge to treat specific devices.

- Verifying A3: CMC tool can provide the appropriate communication occasions.

This assumption is related to “efficient interaction.” The wiki provide the appropriate communication occasions for software development. “Policy for test items (W7)” and “Comment for policy (W8)” of Appendix are the evidences for A3. In W7, a member published the policy for the test item for all the members in the wiki. In W8, Another member in other location replied for the policy by the wiki.

In traditional software development, this communication between members might suspend until regular weekly meeting. In this case, using the wiki provided the appropriate communication occasions and eliminated the delay factor in software development.

## 5 Discussions

We analyzed the case in former section. In this section, we discuss on the effectiveness and efficiency of wiki in the view points of “recording”, “effective knowledge sharing” and “efficient interaction.” We also discuss on the limitation of our analyses based on TIE model for software development communication.

### 5.1 Recording

We discuss the conditions that software developers record the important knowledge in CMC tools. We assume two reasons why they wrote the important knowledge in the wiki.

First reason is that the contents to be shared should be open. The wiki provided the developers with the open communication environment consistently. If they did not use the wiki, they might communicate by face to face meeting, telephone or e-mail. Wiki is more open than these communication methods. The open feature of wiki facilitates developers to write their knowledge. The open feature made casual communication and the developers published the progress information each other. In face to face meeting, powerful members may interfere in the remarks of other low powered members. Wiki may facilitate the remarks of low powered members.

Second reason is that the content in wiki is a single object. In the case, they added the progress information to the document list. If they did not use the wiki, they might share the progress information of each document with e-mail. By e-mail, it is difficult to catch up with the progress information of all the members, because e-mail communication has the feature of cross in the post and makes multi objects to coordinate. The feature of cross in the post of e-mail caused distributes their knowledge. It is not efficient that someone should gather the distributed knowledge by e-mail. The members also need to read all the e-mail to comprehend the all the members' progresses. Because the shared contents feature is open and single object to edit, software developers record the important knowledge in CMC tools.

## 5.2 Effective knowledge sharing

We discuss the conditions that software developers can share the knowledge effectively. We assume that CMC usage is suitable for sharing developers' personal experiences. Particularly, developers' personal experiences are very useful knowledge for the software development with unfamiliar devices. If there is a member who treated with unfamiliar devices, the software development will advance smoothly with the developer's knowledge of unfamiliar devices. As theses kinds of knowledge are not always systematized, the developers can share the knowledge with tacit knowledge network communication. CMC tools facilitate these kinds of knowledge to be share among the members. If this member wrote the knowledge in the wiki in 30 minutes and other eight members read the knowledge 5 minutes, the amount of the time is 70 minuets. On the other hands, if all the nine members acquired the knowledge by try and error in 120 minuets and the amount of the time is 1080 minutes. Although this is an extreme example, knowledge sharing in the CMC tools may be very effective. We assume that sharing unfamiliar knowledge is effective usage of CMC tools.

## 5.3 Efficient interaction

We now discuss the conditions that software developers can interact efficiently. We presume that two imaginary conditions of the software development communication; wiki style and traditional style. We suppose that wiki style has the regular weekly face to face meetings and wiki based communication. Traditional style is assumed to have only the regular weekly face to face meetings.

The relations between the amount of knowledge and time of each style can be described in Fig. 3 based on the above two conditions. Fig. 3 expresses the only the amount of knowledge increased by developers' communication, does not express the amount of the software development works. Straight line in Fig. 3 shows the increase of knowledge in wiki style. The dashed line also shows the increase of knowledge in traditional style.

The amount of increase of knowledge in wiki style is Y(a,b).

$$Y(a,b) = a*wi + b*m \tag{1}$$

In formula (1), a is the number of communication of wiki, wi is the amount of knowledge increase per one wiki communication, b is the number of the face to face meetings and m is the amount of knowledge increase per one meeting.

The amount of increase of knowledge in traditional style is Y(c).

$$Y(c) = c*m \tag{2}$$

In formula (2), c is the number of the face to face meetings and m is the amount of knowledge increase per one meeting. To explain simply, wi and m are fixed in this formula. However, wi and m are variable by the features of communication of wiki and meetings in the real case.

We plot these two formulas in Fig.3 Both wiki and traditional styles have the first meeting in zero point in Fig.3. Both styles gain the same amount of knowledge by first meeting. In wiki style, developers may increase their knowledge to communicate each other three times via wiki. In traditional style, developers may not try to increase the knowledge to communicate each other. Although both styles increase the amount of knowledge by meeting, traditional style may gain the only half amount of the knowledge which wiki style may gain. As a result, wiki style will end the knowledge sharing at point EWi, and its period L may be 3 weeks. Traditional style will end the knowledge sharing at point EM, and its period L+D may be 6 weeks.

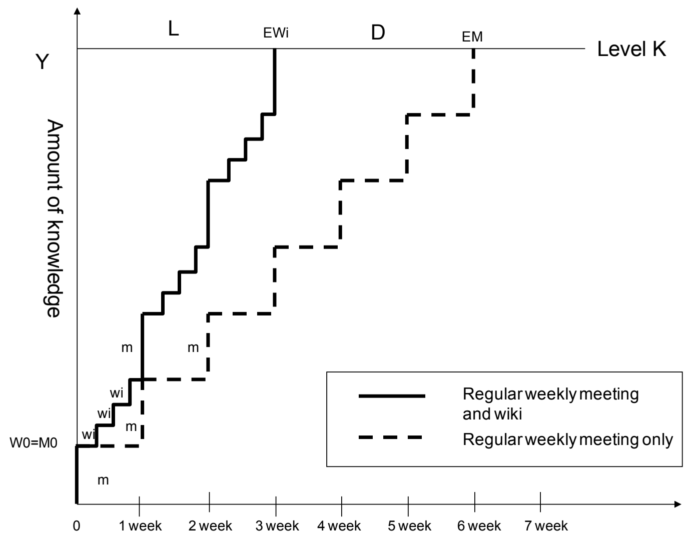


Fig. 3 The relation between time and amount of works

Fig. 3 is the suggestion that effective increases of communication occasions by the wiki cause the knowledge increase. In this suggestion, three interactions to gain the knowledge in wiki match one meeting communication to gain the knowledge. In traditional style, until the development team ends the knowledge sharing, they need seven meetings. In wiki style, they end the knowledge sharing with four meetings and nine interactions in wiki. We suppose the wiki style may be more efficient knowledge communication environment than traditional style.

However, we also found a condition to establish our estimation in Fig.3. The condition is that the knowledge gained by wiki communication should be the same type of knowledge gained by face to face meetings. We supposed that in the real software development field, there are two types of knowledge. One is the knowledge which can be gained by CMC tools such as wiki. Another is the knowledge which can be gained only by face to face meetings. We should distinguish the knowledge to investigate the knowledge communication conditions under which CMC tools such as wiki work adequately.

### 5.4 Limitations

As analyzing the case, we discuss that CMC tools record the important knowledge and make software development process more effective and efficient under the certain conditions. We assure that CMC tools have the ability to facilitate the software development when the development team uses these tools well. In this article, we picked up the few positive effects of CMC tools. We should declare the conditions under which the CMC tools facilitate the software development more circumstantially. Also, we should analyze the negative factors of CMC tools such as false information in the wiki, information overload for developers. And we should investigate the relations among human, CMC content and document, which are the nodes of TIE model.

## 6 Summary

It is important to facilitate the knowledge communication among software developer. We proposed the intermediary knowledge model and TIE model for software development. Traditional software development researches focused on mainly human properties as experts and the quality of documentation. TIE model is software development communication model to explain the just in time documentation. We analyzed the case of the wiki based software development to show the effectiveness and efficiency of the wiki for software development. By analyzing the case, we discussed the conditions to facilitate the software development process by CMC tools. For further study, we should analyze more cases of CMC based software development by using our TIE model. We should also investigate the relation between the communication works and the other important works such as making document or coding.

References

1. Yamamoto, S and Kanbe, M. : Knowledge Creation by Enterprise SNS. The International Journal of Knowledge, Culture and Change Management. Vol.8 No.1 255--264 (2008)
2. Kanbe, M. and Yamamoto, S. : An Analysis of Computer Mediated Communication Patterns. The International Journal of Knowledge, Culture and Change Management. Vol.9 No.3 35--47 (2009)

3. Polanyi, M. : The Tacit Dimension, Routledge & Kegan Paul Ltd., (1966)
4. Nonaka, I. and Takeuchi, H. : The knowledge creating company How Japanese Companies Create the Dynamics of Innovation, Oxford Univ., (1995)
5. Rittel, H. and Kunz, W. : Issues as elements of information systems., Working paper# 1 31. Institute fur Grundlagen der Planung I.A.University of Stuttgart.
6. Conklin, J. and Begeman, M.L. : gIBIS: A Hypertext Tool for Exploratory Policy Discussion., ACM Transactions on Office Information Systems, 4, 6, pp. 303--331 (1988)
7. Conklin J., Selvin A., Shum S., B. and Sierhuis M. : Facilitated Hypertext for Collective Sensemaking: 15 Years on from gIBIS., Hypertext'01 Conference. (2001)
8. Ko, A.J. , DeLine, R. and Venoloa, G. : Information Needs in Collocated Software development teams., 29th International Conference on Software Engineering (ICSE'07) (2007).
9. Ye, Y., Yamamoto, Y. and Nakakoji, K. : Expanding the Knowing Capability of Software Developers through Knowledge Collaboration., International Journal of Technology, Policy and Management Vol. 8, No. 1, pp. 41--58 (2008)
10. Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y. and Asada, M. : Searching the Library and Asking the Peers: Learning to Use Java APIs on Demand., in V. Amaral, L. Veigaet al (eds.): Proceedings of 2007 International Conference on Principles and Practices of Programming in Java, ACM Press: Lisbon, Portugal, pp. 41--50 (2007).
11. Marczak, S., Damian, D., Stege, U. and Schroter, A. : Information Brokers in Requirement-Dependency Social Networks., 16th IEEE International Requirement Engineering Conference, pp. 53--62 (2008)

## Appendix: Contents and intermediary knowledge transformation mode in the wiki in software development process

| Names of pages | ID | Items | contents | Knowledge transformation mode |
|---|---|---|---|---|
| Basic design | W1 | ・Basic design documents List | ・Member added the progress for each items. | Fragmentation, Publication |
| Detail design | W2 | ・Detail design documents List | ・Member added the progress for each items. | Fragmentation, Publication |
| | W3 | ・Tasks | ・Tasks in this process | Publication |
| Make and Unit test | W4 | ・FYI | ・Discussion memo for the decision items | Publication |
| | W5 | ・Policy for test items | ・Descriptions of policy for test items and conditions | Publication |
| | W6 | ・Estimation method of test density | ・Estimation with number of test items and scales | Publication |
| System integration test | W7 | ・Policy for test items | ・Descriptions of policy for test items and conditions | Publication |
| | W8 | ・Comment for Policy | ・Comment for policy of W7 | Collaboration |
| | W9 | ・Estimation method of test density | ・ Estimation with number of test items and scales | Publication |
| | W10 | ・List of the test materials | ・List of the materials; software and hardware | Publication |
| Run time test | W11 | ・Call for comments | ・Message to make the run time test guideline | Publication |
| Development conference #1 | W12 | ・Development conference #1 document list | ・Document list for Development meeting #1 | Fragmentation |
| Development conference #2 | W13 | ・Development conference #2 document list | ・Document list for Development meeting #2 | Fragmentation |
| Graph | W14 | ・the graph description specification | ・Graph specification of system | Publication |
| Know-how | W15 | ・Development Know-how | ・Know-how from the similar system experienced worker | Publication |
| Memo for Project management | W16 | ・Items to be improved for project management | ・Communication method for project management | Publication |
| Demonstration | W17 | ・Purpose | ・Purpose of demonstration | Publication |
| | W18 | ・Scenario | ・Description of use cases for office and factory | Publication |
| | W19 | ・Proposal of the demonstration | ・phases of demonstration and the To Do lists | Publication, Fragmentation, Collaboration |
| Name of Documents | W20 | ・Chapters | ・Chapter and correction comments for documents | Fragmentation, Publication |
| | W21 | ・Documents list | •Documents list and working memos | Fragmentation, Publication |

# Comparison of Coordination Communication and Expertise Communication in Software Development: Their Motives, Characteristics and Needs

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

[1] RCAST, University of Tokyo, Japan
[2] SRA Key Technology Laboratory Inc., Japan
[3] Software Research Associates Inc., Japan

kumiyo@kid.rcast.u-tokyo.ac.jp, ye@sra.co.jp, yxy@kid.rcast.u-tokyo.ac.jp

**Abstract.** Nurturing communication in software development is not about increasing the amount of communication but about increasing the quality of communication experience in the context of software development. Existing studies have shown that different motives and needs are embedded when developers communicate with one another. Identifying *coordination communication* and *expertise communication* as two distinct types of communication, we characterize the difference between the two and discuss important factors to take into account in designing mechanisms to support each type of communication.

**Keywords:** nurturing communication in software development, knowledge collaboration, continuous coordination, unified interface for communication, coordination communication, expertise communication, design considerations

## 1 Introduction

Communication has been taken as an important element in software development. More and more studies argue that socio-technical aspects of software development need to be seriously taken into account in supporting software development. The underlying premise is that peer developers are important knowledge resources in the same way as other artifacts, such as source code, comments, design documents, release notes and bug reports, and that obtaining knowledge and information from their peers is quintessential in software development. Communication should not be regarded as something to get rid of, but instead as something to be nurtured [Nakakoji et al. 2010].

The media currently used in such communication demonstrate a variety of means, including face-to-face, telephone, personal email, mailing-list, Wiki, Internet Relay Chat (IRC), video conferencing, or digital and physical artifacts (e.g., comments inserted in source code or post-it notes pasted on a printed document). Awareness mechanisms may also be regarded as a form of communication media in the sense that one can obtain information about what other members of the projects are doing. As the communication media vary, styles of communications in software development ranges from indirect to direct, from asynchronous to synchronous, and from intentional to unintentional. It might be one to one, one to designated some, or one to unknown numbers of many.

Such peer-to-peer communication in general aims at sharing information and knowledge about a developer's task at hand. By looking into motives of communicative activities of software developers, we have identified two distinctive types of needs in such communications: *coordination communication* and *expertise communicatio*n [Nakakoji et al. 2010].

In coordination communication, a developer tries to coordinate his or her task with the dependent peers in order to avoid and/or to solve emerging or potential conflicts. In expertise communication, a developer seeks for information to solve his or her task at hand and asks his or her peers for help. Note that by expertise communication, we do not mean that a certain group of developers have general expertise therefore they are to transfer their knowledge to novice developers through communication. In contrast, our view is that expertise is always defined in terms of some context, for instance, in terms of a particular method, of a particular class, of a particular release, or of a particular bug report at a particular point in time; and that expertise is not something definable without context. With this view, each developer has his or her own expertise in some aspects of the system and the project. Expertise communication, therefore, may take place among all members of the peer developers in every direction [Ye et al. 2008].

Developers currently do not distinguish the two types of communication, which are driven by their “information needs” and are carried out through common communication channels. Coworkers were the most frequent source of information by software developers and that two most frequently sought information by software developers that depended on their coworkers were “what have my coworkers been doing?” and “in what situations does this failure occur?” [Ko et al. 2007]. The former information is sought primarily for the purpose of coordinating the work and the latter is for the purpose of getting some knowledge about the source code. Data on three well-known open source projects have shown that text-based communication (mailing lists and chat systems) are the developers’ primary source of acquiring both general knowledge about other developers (to know who has necessary expertise) and specific awareness of who is working on their relevant parts of the system (to coordinate their tasks) [Gutwin et al. 2004].

Developers often mix the two types of communication within a single discourse session without paying an attention to distinguish the two. For instance, a developer John first asks his colleague Mary over the cubicle wall if she knows why the class C calls a method X instead of Y, then Mary answers that it is because Y is planned to be thrown away, and that by the way Mary has just been working on X and checked-in the changes therefore he had better check the latest version of X if he is working on C.

Thus, while the initial question posed by John is expertise communication (i.e., he wanted to ask Mary to give him the answer as to why C calls X instead of Y), the subsequent conversation provided by Mary turns out to be coordination communication (i.e., C that John is working on depends on X that Mary is working on).

Why does it matter then to distinguish the two types of communication if developers do not distinguish the two? It matters because when it comes to design computational mechanisms for supporting communication in software development, each type of communication demands different types of concerns.

This position paper first describes what fundamental differences exist between the two types of communication in software development. We then explain how different aspects need to be considered in designing computational support mechanisms. We conclude with a list of research issues in developing such support.

## 2 Expertise communication and Coordination Communication

A few distinctive features are involved in each of expertise communication and coordination communication.

Let us first illustrate coordination communication. Suppose a developer X initiates communication with another developer Y, which turns out to be coordination communication. The purpose of the coordination communication is to coordinate tasks to resolve emerging conflicts or to avoid possible future conflicts among the tasks X and Y are engaged in. X and Y are called "socially dependent developers" [de Souza et al. 2007] in the sense that they have to coordinate their tasks through social interactions when the perceived conflicts become necessary to be resolved. X and Y together form an "impact network" [de Souza et al. 2008]. Coordination communication is a part of impact management, which is "the work performed by software developers to minimize the impact of one's effort on others and at the same time, the impact of others into one's own effort" [de Souza et al. 2008]. X may need to further involve those developers who are part of the impact network.

In contrast, suppose a developer A initiates communication with another developer B, which turns out to be expertise communication. The purpose of the expertise communication is for A to get some information about A's task at hand. A is asking B to help A by providing some information for A's particular task. As noted above, we use expertise communication to refer to the kind of activities by seeking information that is essential to accomplish A's software development activities, not for the purpose of learning, but for the purpose of performing A's job. If A does not get satisfying information from B, A might need to ask other peers for the same question.

Thus, while the relation between X and Y in the coordination communication is reciprocal, that of A and B in the expertise communication is not. In coordinating communication, there is a symmetric or reciprocal relation between those who initiate communication and those who are asked to communicate with roughly equal interests and benefits. In expertise communication, in contrast, there is an asymmetric and unidirectional relation between the one who asks a question and the one who is asked to help. The benefit would primarily for the communication initiator and the cost (i.e., additional efforts) is primarily paid by those who are asked to participate in the communication; that is, the cost of paying attention to the information request, of stopping his or her own ongoing development task, of composing an answer for the information-seeking developer while collecting relevant information when necessary, and of going back to the original task [Ye et al. 2007].

The role and value of resulting communicative actions would also differ between the two types of communication. When developers communicate with one another, their conversations as well as produced artifacts (mail message contents or white board drawings, for instance) can be stored (if appropriate media is used).

Such recorded communication can be useful if generated through expertise communication. Email exchange about a particular design of a class, for example, would serve as a valuable auxiliary document for the class, and another developer might find it useful to read when using the class at a later time.

Archived communication generated through coordination communication might be useful to inform other developers within the same impact network for the time being. However, the impact network constantly changes over time and such information communicated over a particular class would soon become obsolete. Moreover, coordination communication without its temporal context could be quite harmful when misused. A collection of the coordination communication about a particular object over a long period of time may serve as the object's development log but it would not be more than the existing developmental records captured within current development environments.

Table 1 summarizes the differences of coordination communication and expertise communication.

Table 1: Comparing Coordination Communication and Expertise Communication

| | **Coordination Communication** | **Expertise Communication** |
|---|---|---|
| *purpose* | to coordinate work | to get information |
| *needs* | conflict avoidance, conflict resolution | problem solving |
| *cost & benefit* | reciprocal between a communication initiator and the other communication participants | asymmetric between a communication initiator and the other communication participants |
| *expanding participants* | when others are part of the impact network | when the initiator could not get satisfying information |
| *recorded communication* | useful for the time-being until the impact network changes | become valuable document for later use |

The next section compares the different aspects of concerns in designing mechanisms for supporting each type of the communication.

## 3. Different Needs for Supporting the Two Types of Communications

"*A thing is available at the bidding of the user--or could be--whereas a person formally becomes a skill resource only when he consents to do so, and he can also restrict time, place, and method as he chooses*" [Illich 1971].

In talking about depending on teachers as knowledge resources, Illich argued that their willingness to participate is essential in regarding people as information resources. Using peers as potentially relevant information resources is likely to increase cognitive load for both of those who initiate communication, and those who are asked to participate in the communication. Unlike the Help Desk where the jobs of those who are asked is to answer [Ackerman et al. 1990], peer developers are not there to communicate but to perform their own development tasks in a time-critical fashion. They might be willing to communication if they had more time and less stressful situations; they might not otherwise unless they see immediate needs for themselves to communicate.

Therefore, asymmetric nature of the beneficiary and benefactors in expertise communication demands a critical attention in designing communication support

Table 2: Different Present Research Emphases on the Two Types of Communication

| | **Coordination Communication** | **Expertise Communication** |
|---|---|---|
| *key concepts* | continuous coordination [Redmiles et al. 07]<br>impact management [de Souza et al. 08] | developer as knowledge resources [Nakakoji 06]<br>communication channel [Ye et al. 07] |
| *primary functionality* | awareness<br>visualization | finding expertise<br>choosing experts<br>socially-aware communication channel |
| *tools* | Palantir [Sarma et al. 03]<br>Ariadne [de Souza et al. 07] | Expert Finder [Vivacqua et al. 2000]<br>Expertise Browser [Mockus et al. 2002]<br>STeP_IN_Java [Ye et al. 2007] |
| *socio-technical aspects* | social interaction needs are inferred from the technical (structural) dependencies of the tasks [Herbsleb et al. 1999] | communication participants are selected based on their technical experiences on sought information and previous social relations with an information seeker [Ye et al. 2007] |

mechanisms. For an information-seeking developer, involving more participants in the communication means having more potential information resources, implying a better chance of obtaining necessary information at the cost of information overload; thus high quality ranking and triaging mechanisms would become essential. For those who are asked to participate in the communication and provide information, however, responding to the request becomes yet another task [Ye et al. 2007].

On one hand, when the relation between the communication initiator and the rest of the communication participants is symmetrical and reciprocal, those who are asked to participate in the communication would feel the equal importance of engaging in the communication. On the other hand, when the relation is asymmetrical where the initiator would be a beneficiary and the other participants would be benefactors, mechanisms to persuade people to participate in the communication are necessary.

Although there had been no explicit distinctions of the two types of communications in software development, existing research currently demonstrates different emphases on supporting each aspect of the communication with regard to key concepts, tools, and the primary functionality. Both approaches stress the importance of taking socio-technical aspects into account, but in different contexts.

Table 2 illustrates the two distinctive approaches.

Supporting coordination communication has been primarily studied in such research areas as coordinating programmers and programming tasks. Supporting expertise communication has been primarily studied in such research areas as knowledge sharing and experts finding.

Although they do not explicitly use the term coordination communication, Redmiles et al. [2007] present the continuous coordination paradigm for supporting coordination activities in software development. The paradigm contains with the following four principles: first, to have multiple perspectives on activities and information; second, to have non-obtrusive integration through synchronous messages or through the representation of links between different sites and artifacts; third, to combine socio-technical factors by considering relations between artifacts and authorship so that distributed developers can infer important context information; and fourth, to integrate formal configuration management and informal change notification via the use of visualizations embedded in integrated software development environments [Redmiles et al. 2007].

This paradigm stresses the importance of integrating the coordination activities within the programming environment, and of making developers aware of the need of communication and simultaneously minimizing the distraction of software developers by using formal configuration management mechanisms and informal visual notification and awareness techniques. They focus on socio-technical factors in the sense that peer-to-peer coordination communication needs are inferred by analyzing structural (technical) dependencies of the system components they are working on because they have to coordinate their tasks through social interactions when the perceived conflicts become necessary to be resolved [Wagstrom et al. 2006; de Souza et al. 2008].

Nakakoji et al. [2010] present nine design guidelines for expert communication support mechanisms. The guidelines state that expert communication support mechanisms should: be integrated with other development activities; be personalized and contextualized for the information-seeking developer; be minimized when other types of information artifacts are available; take into account the balance between the cost and benefit of an information seeking developer and the group productivity; consider social and organizational relationships when selecting developers for communication; minimize the interruption when approaching those who are selected to be communicated with; provide ways to make it easier for developers to ask for help; provide ways to make it easier for developers to answer or not to answer for the information requested; and be socially aware.

The guidelines stress the importance of finding communication participants who not only have necessary information, but are also willing to provide the sought information in an appropriate way in a timely manner. The guidelines also pay an attention to the cost of those who are asked to engage in expertise communication, and argue for the use of socially-aware communication channels. They focus on socio-technical aspect in a sense that finding potential communication participants takes into account not only technical skills of developers but also their social relationship with the information-seeking developer.

Such differences of the two types of communication necessitate fundamental differences in designing communication support mechanisms in:

- how to select who to participate in communication,
- in what timing to start communication,
- how to invite people to participate in the communication,
- through which communication channel, and
- how to use the resulting communicative session (i.e., communication archives).

Table 3 lists factors that are common and distinctive to the two types of communications in software development.

Table 3: Comparison of Design Factors

| | **Coordination Communication** | **Expertise Communication** |
|---|---|---|
| *in relation to the development environment* | integrate with the development environment | |
| *disturbance* | minimize | |
| *communication needs are identified when:* | conflicts are detected or possible conflicts are detected | a developer is in need of information about the task at hand |
| *trade-off of not communicating* | potential risks of conflicts that might arise by not coordinating | potential risks of problems when appropriate information is not provided to the information-seeking developer |
| *alternative means to minimize communication* | to visualize the status of the potential conflicts so that a developer may not need to engage in explicit communication by glancing at the visualized information | to guide the information seeking developer to relevant artifacts such as source code and documents so that a developer may not need to engage in explicit communication |
| *the use of the object a developer works on* | by looking at what objects a developer presently works on in order to infer the impact network | by looking at what objects a developer previously worked on in order to infer the technical expertise of the developer |
| *the use of who is initiating the communication* | by using the communication initiator's impact network in selecting communication participants | by using the communication initiator's social relations in selecting communication participants |
| *helping one in initiating communication* | mechanisms to switch to an explicit communication mode with the peers in the impact network when urgent communication needs are detected | mechanisms to ask without worrying about bothering peers |
| *helping those who are asked to participate in the communication* | mechanisms to judge how urgent and important the conflict is | mechanisms to minimize feeling guilty not responding to the request |
| *communication channel needs to be:* | impact-aware so that developers can easily judge and communicate how much impact the emerging conflict might have and how to avoid and solve the conflict. | socially-aware so that developers use the right channel instead of the channel that is easier to use (whom to ask, through which media) |

Figure 1 illustrates how communication support mechanisms should be built in support of software developers. On one hand, there should be a unified interactive framework with communication for a software developer that is integrated within a development environment. They should not be needed to explicitly choose which communication type they would like to be engaging in. Communication with peer developers should be supported as another type of information usage during software development, and needs to be integrated with a program- and document-authoring and browsing environment. On the other hand, how the communication is designed and structured needs to be tuned to each of the two types of communication. What is needed is to take the above differences seriously into account and design the communication support mechanisms accordingly.

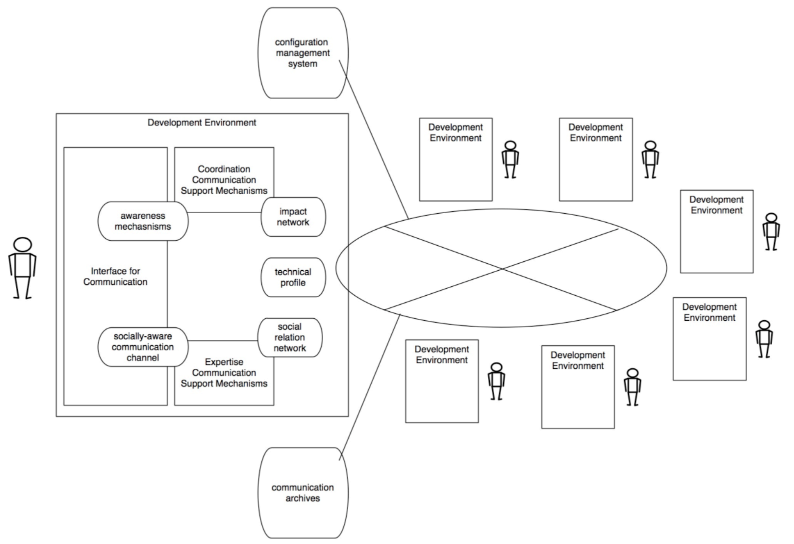


Figure 1: An Architecture of Communication Support Mechanisms that Takes Into Account Two Types of Communication

## 4. Concluding Remarks

Nurturing communication in software development is not about increasing the amount of communication but about increasing the quality of communication experience in the context of software development. Although having been recognized merely as communicative acts, different motives and needs are embedded when developers communicate with one another. Different computational mechanisms are necessary to realize successful communication. This paper presents our initial attempt to list different aspects necessary to take into account in designing mechanisms to support each of the coordination communication and expertise communication. There are no general communication needs but either coordination communication needs or expertise communication needs. A real challenge would be to design a developer-centered unified interactive framework that seamlessly integrates the two.

## References

[Ackerman et al. 1990] Ackerman, M. S., T. W. Malone, Answer Garden: A Tool for Growing Organizational Memory. Proceedings of the ACM Conference on Office In-formation Systems. Cambridge MA: 31-39, 1990.

[de Souza et al. 2007] de Souza CRB, Quirk S, Trainer E, Redmiles D: Supporting collaborative software development through the visualization of socio-technical dependencies. In: Proc. of GROUP'07, pp 147–156, 2007.

[de Souza et al. 2008] de Souza CRB, Redmiles D: An empirical study of software developers management of dependencies and changes. In: Proc. of ICSE'08, pp 241–250, 2008.

[Gutwin et al. 2004] Gutwin C, Penner R, Schneider, K., Group awareness in distributed software development, Proceedings of the 2004 ACM conference on Com-puter supported cooperative work, 72-81, 2004.

[Herbsleb et al. 1999] Herbsleb, J., Grinter, R. E. Splitting the Organization and Integrating the Code: Conway's Law Revisited. Proceedings of International Conference on Software Engineering (ICSE99): 85-95, 1999.

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

[Ko et al. 2007] Ko AJ, DeLine R, Venolia G: Information needs in collocated software development teams. In: Proc. Of ICSE’08, pp 344–353, 2007.

[Mockus et al. 2002] Mockus A, Herbsleb J: Expertise Browser: A quantitative approach to identifying expertise. In: Proc. of ICSE'02, pp 503–512, 2002.

[Nakakoji 2006] Nakakoji K: Supporting software development as collective creative knowledge work. In: Proc. of KCSD2006, Tokyo, pp 1–8, 2006.

[Nakakoji et al. 2010] Nakakoji, K., Ye, Y., and Yamamoto, Y.: Supporting expertise communication in developer-centered collaborative software development environments. In: Finkelstein, A., van der Hoek, A., Mistrik, I., Whitehead, J. (eds) Collaborative Software Engineering, Springer-Verlag, 2010 (forthcoming).

[Redmiles et al. 2007] Redmiles D, van der Hoek A, Al-Ani B, Hildenbrand T, Quirk S, Sarma A, Filho RSS, de Souza C, Trainer E: Continuous coordination: a new paradigm to support globally distributed software development projects. Wirtschaftsinformatik J, 49: S28–S38, 2007.

[Sarma et al. 2003] Sarma, A, Noroozi Z, van der Hoek, A,: Palantir: raising awareness among configuration management workspaces. In: Proc. of ICSE'03, pp 444–454, 2003

[Vivacqua et al. 2000] Vivacqua A, Lieberman H: Agents to assist in finding help. In: Proc. of CHI'00, pp 65–72, 2000

[Wagstrom, et al. 2006] Wagstrom, P. and J. Herbsleb, Dependency Forecasting, CACM 49(10): 55-56, 2006.

[Ye et al. 2007] Ye Y, Yamamoto Y, Nakakoji K: A socio-technical framework for supporting programmers. In: Proc. of ESEC/FSE'07, pp 351–360, 2007.

[Ye et al. 2008] Y. Ye, Y. Yamamoto, K. Nakakoji, Expanding the Knowing Capability of Software Developers through Knowledge Collaboration, IJTPM (International Journal of Technology, Policy and Management), Special Issue on Human Aspects of Information Technology Development, Inderscience Publishers, Vol.8, No.1, pp.41-58, 2008.