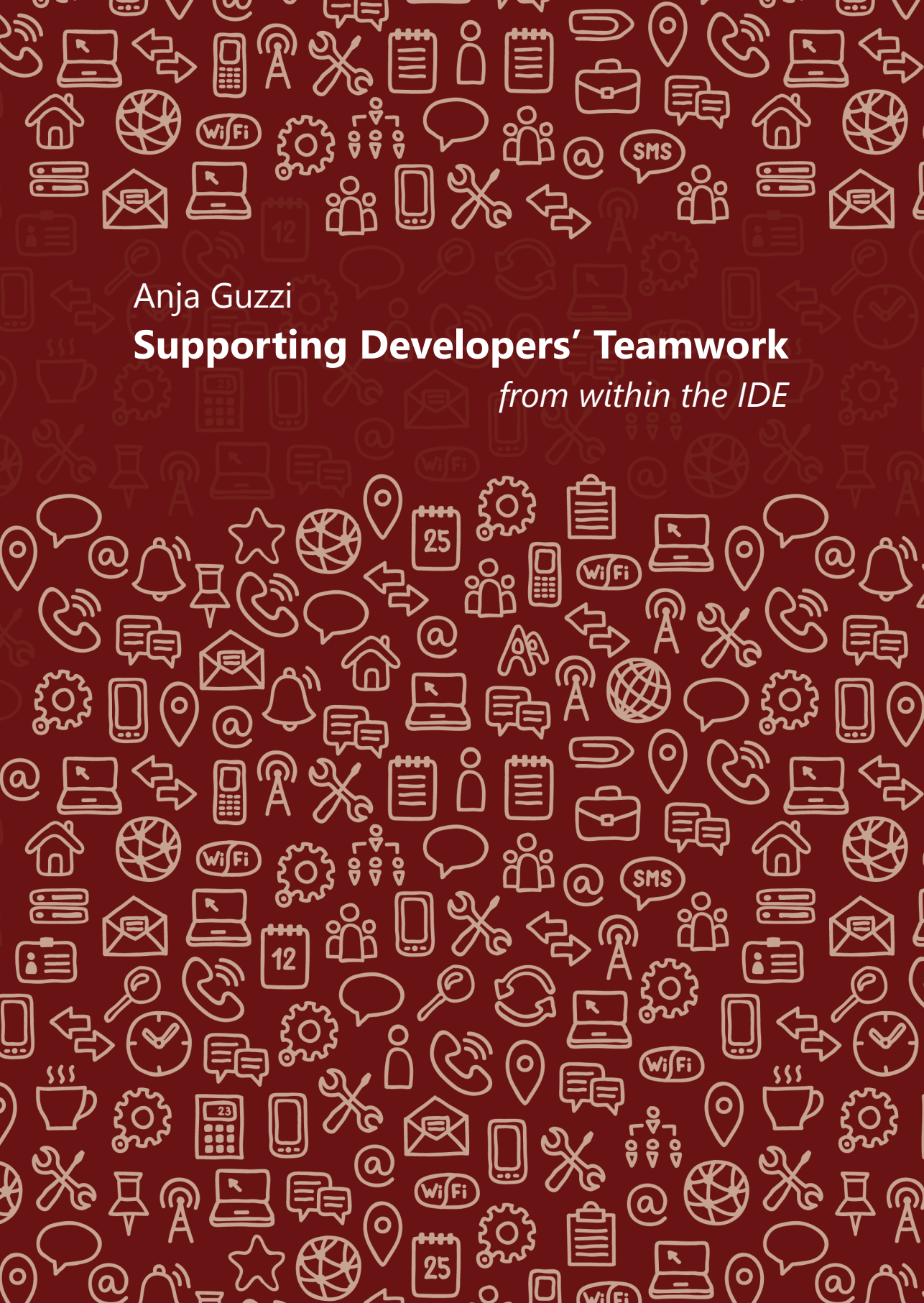


Anja Guzzi

Supporting Developers' Teamwork

from within the IDE

Supporting Developers' Teamwork

from within the IDE

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus Prof. ir. K.C.A.M. Luyben, voorzitter van het College voor Promoties, in het openbaar te verdedigen op maandag 30 maart 2015 om 12.30 uur door

Anja GUZZI

Master of Science in Informatics
geboren te Faido, Zwitserland.

This dissertation has been approved by the promotors:

Prof. dr. A. van Deursen	Delft University of Technology, The Netherlands
Prof. dr. M. Pinzger	University of Klagenfurt, Austria

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. A. van Deursen	promotor
Prof. dr. M. Pinzger	promotor
Prof. dr. ir. D.M. van Solingen	Delft University of Technology, The Netherlands
Prof. dr. A. Hanjalic	Delft University of Technology, The Netherlands
Prof. dr. P. Lago	VU University Amsterdam, The Netherlands
Prof. dr. M. Lanza	University of Lugano, Switzerland
Dr. R. DeLine	Microsoft Research, Redmond, USA

The work in this dissertation has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). IPA thesis number 2015-05.

This research has been financially supported by the Netherlands Organisation for Scientific Research project 612.001.019: *What your IDE could do once you understand your code*.

ISBN 978-94-6186-435-2

Author email: anja.guzzi@gmail.com

Contents

Part I

Overture

1 Introduction

Software engineering is a team effort: Sometimes even hundreds of professionals collaborate to devise, build, evaluate, and later evolve a software system. However, developers working together face many issues that hinder teamwork: For example, communication in natural language is ambiguous, human memory cannot remember all of the project's details, and keeping track of what everyone is doing even in a small group is cumbersome. As a result, teamwork in software engineering is time-consuming and problematic.

Given that most of developers' time is spent within the Integrated Development Environments *(IDE), researchers have started tackling difficulties of teamwork in software engineering by adding collaborative capabilities to such environments. Nevertheless, the IDE remains mostly a soloist tool that primarily helps individual programmers to be more effective during the classical edit-compile-run cycle.*

With this dissertation, we explore how to better support developers' teamwork from within the IDE.

1.1 Background and Motivation

Teamwork is fundamental in the production of software. Ever since the first development company was founded, the creation of software has been the result of team effort [106]. Due to the growing complexity of software systems and the urge for faster time-to-market, over the years teamwork has become more and more fundamental to the production of software [80]. Nowadays, having a large development team to create software is the norm rather than the exception.

Effective teamwork requires successful coordination and collaboration [105]. The terminology used in many disciplines [115] defines coordination as "*managing dependencies between activities,*" and collaboration as "*peers working together.*" Despite the clear advantages of successful teamwork, the difficulty in coordinating developers and teams and make them collaborate is one of the main reasons why the software industry has been in crisis since its inception [105]. Multiple studies have provided evidence of the plethora of problems emerging in software development when teamwork is not successful. For example, among others, Sarma *et al.* presented how a decreased communication between team members working on related artifacts is connected to defects and software failures [150], Grinter [67] and Herbsleb and Moitra [86] reported that flawed team allocation may lead to low willingness to help out other team members, and Nagappan *et al.* found a strong relationship between an overly hierarchical team structure and post-release defects [124].

An ample body of research in supporting the coordination of developers and teams has shown that it is "one of the most difficult-to-improve aspects of software engineering" [14]. In fact, team effort in software development spans different aspects, such as facing the challenge of understanding a program written by someone else and a long time before (*i.e., program comprehension* [22]), maintaining effective communication between dependent engineers [81], dealing with past design decisions that materially affect today's work [110], and coordinating shared goals, priorities, responsibilities, and schedules [137].

Additionally, conducting software projects with teams in more than one location, often in different continents (also known as global software engineering [89]) is not unusual [87] and exacerbates the difficulties of handling coordination and collaboration [86, 26], by posing, for example, cultural barriers, process issues, and different communication needs.

The computer-supported collaborative work (CSCW) [66] community is focused on coordinated activity that can be computer-assisted and is carried out by groups of collaborating individuals [7]. There is a branch of CSCW research devoted to support developers' teamwork, which, over the years, has developed theories on teamwork for software projects (*e.g.,* [88, 27]) and has proposed a number of solutions to the practical challenges faced during collaborative software development (*e.g.,* [31, 62]). For example, researchers proposed systems to help manage the team development process, by supporting managers and developers in assignment of work, monitoring progress, and improving processes [20, 174].

Many of the proposed solutions are based on the design and development of *groupware* [95], which is "software that accentuates the multiple user environment, coordinating and orchestrating things so that users can "see" each other, yet do not conflict with each other" [114]. In particular, given the large amount of time developers spend in the *integrated development environment* (IDE) [110], a number of approaches have been proposed to support developers' teamwork in the IDE. Prominent examples are Jazz [92] and Mylyn [54]: full-fledged platforms built on top of the IDE aimed at transforming the IDE into a comprehensive collaboration tool. With respect to the approaches for supporting teamwork, Sarma *et al.* present a comprehensive review of coordination tools and defines a framework that classifies those technologies according to multiple coordination paradigms (such as communication, artifact management, and task management) [152].

CSCW research applied to software engineering has to take into account that developers' teamwork involves "developing shared understanding surrounding multiple artifacts, each artifact embodying its own model, over the entire development process" [176]. As a result, teamwork research in software engineering cannot be artifact-neutral, rather it has to take into account the artifacts (mainly the source code) being created and evolved and their embedding in the development process at large.

The concerns we seek to address in this work are that (1) many of the artifact-aware proposed approaches to support teamwork are heavyweight, thus may disrupt the development workflow, habits, and development process in place in a team; and (2) such approaches are often targeted to a specific type of teams (*e.g.*, very large teams in industrial context, with a long development history) or require a steep learning curve.

Therefore, we aim at supporting teams independently from the development process they have in place: For example important improvements in their teamwork have been achieved by projects implementing agile software development methods [117], such as SCRUM [157] that propose a flexible, iterative, and early-feedback based approach to manage collaborative software development. Our goal is to devise solutions that do not disrupt a development process already in place, but possibly complement it. Moreover, we aim at improving both distributed and co-located settings. In fact, we argue that appropriate teamwork solutions can be beneficial to both distributed and co-located teams. In the former, the space for improving coordination is greater, because "distance matters" [130] and disrupts many mechanisms that function to coordinate the work in co-located settings [87]; in the latter the actual distance might be different from the perceived ones [140] (a door closed or a different floor, can wipe out the benefits of being co-located [89]), thus the advantages of appropriate teamwork support can become unexpectedly important.

Overall, our research goal is to devise software-artifact-aware lightweight IDE additions that can be seamlessly integrated in the development workflow of a variety of teams and that provide additional effective support to developer's teamwork, while requiring little learning time.

1.2 Research Questions

We build our work to support developers' teamwork in the IDE with lightweight additions around four research questions, which we devised and refined through the years leading to the completion of this dissertation. We follow the advice of Johansen on the creation of groupware: "A good rule of thumb is to look for current pains or problems [...] to identify those things that must be done right for an organization to succeed" [95], thus we start by investigating the support that is still needed in today's development practice and the leeway for improvement. Based on this, we phrase our first research question:

[1] How do developers experience collaboration in teamwork?

While answering this question, our results highlighted that developers' needs in teamwork mostly regard coordination, which requires a substantial amount of information to be shared among team members. To achieve successful teamwork practice it is therefore important that this information (which usually resides only in the head of few developers) is visible and easily accessible to the team. Our results, thus, reiterate on the concept of *information sharing*, which has been put in evidence as a key point for successful team coordination by several studies (*e.g.,* [26, 89, 45]) not only in software engineering (*e.g.,* [93, 120]). This need for information sharing also closely relates to the need for *awareness*—"an understanding of the activities of others, which provides a context for your own activity" [50], and several researchers presented techniques to help developers maintaining awareness during the development process (*e.g.,* [91, 24]).

Before diving into investigating approaches and solutions to tackle the retrieval and display of useful information from existing recorded data and the creation of approaches to record and share novel information, we also explored the special case of open-source software development, to derive useful information to guide our research and devise our approaches. On this, we formulate our second research question:

[2] How is information shared in open source software projects?

After we set the ground for information sharing needs, we tackle two complementary aspects of information sharing in teamwork: Exposing information that is already recorded (*e.g.,* by version control systems) and support the creation of new useful information (*e.g.,* for program comprehension). In fact, our results also showed that developers struggle with certain development scenarios when information is lacking and that, although part of the necessary information could be retrieved from the traces developers leave in software repositories [84], this does not happen in practice. Moreover, a significant portion of information that would be useful (*e.g.,* the rationale of a change and past program comprehension efforts) is not recorded as there is neither good motivation nor a simple way for developers to record and share it. We structure our effort along two research questions, trying to devise approaches applicable to most types of development teams and processes:

[3] How can we expose existing information to support teamwork?

[4] How can we aid information creation to support teamwork?

Table 1.1: *Mapping of research questions to chapters*

Research question		Chapters
[1]	How do developers experience collaboration in teamwork?	2
[2]	How is information shared in open source software projects?	3
[3]	How can we expose existing information to support teamwork?	4 and 5
[4]	How can we aid information creation to support teamwork?	6 and 7

1.3 Research Outline

Table 1.1 illustrates how the research questions are addressed in the different chapters of this work. In the following we outline in mode detail the different parts that compose this dissertation.

Part I: Prologue

In this first part of the dissertation, we first introduced the background and stated the research goal for our work, then we presented the central research questions we answer to approach our goal. After that, we illustrate the research method we follow. Then, in the following chapters of this part, we present: (1) An exploratory study we conduct to answer our first research question and that sets the empirical basis of the work that we present in the rest of the dissertation, and (2) an analysis of how teamwork takes place in open-source software (OSS) projects, particularly by focusing on the analysis of the development mailing list, considered the hub of OSS project communication.

Chapter 2 provides a background exploratory work aimed at understanding how developers experience teamwork and at discovering relevant and actionable problems developers face when working in team. By reflecting on the outcome of this work, we want to understand the leeway to introduce effective lightweight improvements in the IDE. We find that developers' needs in teamwork mostly regard managing dependencies between activities, rather than working together concurrently on the same (sub)task. Central in our study is a renewed emphasis on the importance of *shared information* as a key enabler for developers' teamwork. We use this concept of shared information in the remainder of the dissertation to address our research goal of supporting developers' teamwork *from within the IDE* (*i.e.,* by creating lightweight software-artifact-aware extensions of the IDE).

Since our study participants are mostly working in the context of industrial development, in which team members have clear duties and roles, we also analyze the OSS context. This helps us understanding how to be more effective in this scenario and gives further motivation for our approaches.

Chapter 3 presents an extensive qualitative assessment of the development mailing list of Lucene, a mature and widely used OSS system. Our aim is to update our knowledge of communication and teamwork in open source settings. We find that core developers are involved in less than 75% of the technical discussions. Furthermore, we find that development emails are losing their role as the hub of project communication and that other channels, such as issue repositories, are gaining more popularity. This increased popularity seems to be mostly due to the closer connection between the communication means and the software artifacts; this strengthens the motivation for targeting teamwork solutions in the IDE—the place in which developers work with these artifacts.

Part II: Exposing Information

In the second part of our work, we cover one side of shared information for teamwork: Exposing latent, but *existing* information to developers in order to support coordination and program comprehension. To this aim, we propose two lightweight approaches. The first approach proposes a domain-specific, IDE-embedded, photo-oriented communication tool, and the second approach critiques and rethinks the current support for receiving changes in the IDE and uses the result of the first approach to display relevant information. In this part, we address our second research question.

Chapter 4 presents a year-long series of surveys and interviews that we conduct to better understand how and why software developers discover and communicate with one another. Based on these results, we design and implement a tool, CARES, to encourage developers' communication with one another and to simplify the process of doing so. CARES displays a context-sensitive array of photos of the engineers who are most tightly connected to the code in each file currently being edited in the IDE, lets a developer make an informed choice about whom to contact, and enables the communication with them. By deploying our tool at a large US-based company, we find that most users report that it simplifies the process of finding and reaching out to other developers and offers them a sense of community with their colleagues, even with those colleagues not currently working on their team.

Chapter 5 presents an analysis of the current IDE support for receiving code changes, which is the most problematic scenario emerged from our initial exploratory work (Chapter 2). Analyzing IDE support in the light of widespread usability heuristics, we find that it does not properly support the developers' information needs on change history. Based on this, we derive requirements for a lightweight IDE extension, we provide the design of a tool, BELLEVUE, to realize these requirements, and we evaluate it with senior developers. Developers report that BELLEVUE has powerful features and that they would use it frequently in their daily activities.

Part III: Creating Information

In the third part of our work, we cover the other side of shared information for teamwork: To aid developers creating *new* information that, if shared, can support teamwork. Information sharing often does not happen autonomously because developers should dedicate extra effort and time without seeing immediate benefits. Our idea is to provide quick and simple mechanisms that require little time and effort to share information and that can also immediately benefit the information creator. In particular we offer support to the developers working on a specific task and who often have valuable information residing only in their head, to create new information with the goal of helping themselves and future developers conducting program comprehension tasks. We propose two approaches with increasing complexity, yet lightweight, implemented as IDE extensions: (1) an improved code bookmarking mechanism that support sharing, and (2) a short text messaging system, automatically enriched with collected interaction data. In the former, the incentive for sharing information derives from the fact that the bookmarking activity is done for oneself, and the information sharing is a byproduct of this; in the latter, the incentive for sharing information derives from the possibility to let other people know about one's status. In this part of our work, we address our third research question.

Chapter 6 presents an online survey and interviews with professional software engineers about their current usage and needs of code bookmarks, and the design and implementation of a tool, Pollicino, for collective code bookmarking. Based on our results, we devise Pollicino to encourage developers to bookmark artifacts while investigating the source code and to document their findings with a description associated to a bookmark, which can be shared with other team members. Conducting an experiment with developers using Pollicino, we find that it can be used effectively to (micro-)document developers' findings, which can be later used by others in their team.

Chapter 7 presents James, an approach to extend the IDE with a (Twitter-like) micro-blogging facility. We encourage developers to share the information they build up while understanding a piece of code, so that it does not evaporate after the corresponding task is finished. James lets developers write status updates on their tasks and combines it with interaction data automatically collected from the IDE. By conducting an empirical evaluation of the proposed approach, we get initial evidence that developers are willing to micro-blog on their activities and that the combined interaction and micro-blogging data is helpful in maintenance tasks.

Part IV: Epilogue

In the last part of our work, we take a step back from the different research questions and studies done. We consider our work as a whole, we present our contributions, and we outline future research directions we envision.

Chapter 8 concludes this dissertation by discussing our approach and findings, providing answers to our research questions, summarizing the contributions of this work, and outlining future research directions.

1.4 Research Method

To conduct our research we use a variety of research methods. We often use a mixed method approach [36] in order to obtain new insights, or to *triangulate* common findings, using different methods. In this way, we acquire a better understanding of the research problems than when using one approach alone. Since all research questions address people and the way they collaborate in teams, we adopt a range of qualitative methods also commonly used in the social sciences, such as grounded theory, interviews, surveys, and case study research [55]. We also follow software repository mining approaches [98]. More details on the research methods of the different studies are provided in each chapter.

Our research is based on two pillars: involving practitioners and devising tools. Whenever possible we follow the full path from exploring a teamwork problem by understanding current practices (*i.e.,* doing direct data collection [111] by involving developers through interviews and surveys), through the design/implementation of a tool to solve teamwork issues, to the evaluation of the solution with users. This allows us not only to verify the effectiveness of the proposed solutions, but also to focus on the most critical issues developers have to face in their daily work. When appropriate (*e.g.,* to study communication habits in open source software projects), we also conduct indirect data collection [111] by analyzing the documents written by practitioners and users of a software project.

To encourage repeatability and further research on the same topics, unless confidential because of the industrial context, we share online the tools we implemented, the data we collected (both in raw and processed format), and the notes we took during our studies. All the available material can be found at `http://www.st.ewi.tudelft.nl/~guzzi/`.

1.5 Origin of Chapters

The main chapters (Chapters 2–7) of this document are small adaptations of papers published or submitted to conferences. Since the chapters are small adaptations of the papers which were written separately, they can be read independently of each other, at the price of some redundancy in background, motivation and examples. The author of this dissertation is the main author of all the publications included as chapters.

- Chapter 2 is the first part of the paper[1] "Supporting Developers' Coordination in The IDE" [74], accepted for publication in the proceedings of the 18th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2015). The authors of this publication are Guzzi, Bacchelli, Riche, and van Deursen.

- Chapter 3 contains our paper "Communication in OSS Development Mailing Lists" [73], published in the proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013). The authors of this publication are Guzzi, Bacchelli, Pinzger, Lanza, and van Deursen.
- Chapter 4 is a blended version of the paper "Facilitating Enterprise Software Developer Communication with CARES" [76], published in the proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), with authors Guzzi, Begel, Miller, and Nareddy; and the paper "Facilitating Communication between Engineers with CARES" [75], published in the companion (tool demonstration track) proceedings of the 34th International Conference on Software Engineering (ICSE 2012), with authors Guzzi and Begel.
- Chapter 5 is the second and last part of the paper[1] "Supporting Developers' Coordination in The IDE" [74], accepted for publication in the proceedings of the 18th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2015). The authors of this publication are Guzzi, Bacchelli, Riche, and van Deursen.
- Chapter 6 contains our paper "Collective Code Bookmarks for Program Comprehension" [77], published in the proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC 2011). The authors of this publication are Guzzi, Hattori, Pinzger, Lanza, and van Deursen.
- Chapter 7 is an extended version of the paper "Combining Micro-Blogging and IDE Interactions to Support Developers in their Quests" [79], published in the proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010). The authors of this publication are Guzzi, Pinzger, and van Deursen.

Apart from these publications the author of this dissertation has (co-)authored the following publications, which have been created during the PhD trajectory but are not directly included in this thesis:

- "Documenting and Sharing Knowledge about Code" [72], published in the companion (doctoral symposium track) proceedings of the 34th International Conference on Software Engineering (ICSE 2012). The sole author of this publication is Guzzi.
- "Adinda: A knowledgeable, Browser-Based IDE" [171], published in the companion (NIER track) proceedings of the 32nd International Conference on Software Engineering (ICSE 2012). The authors of this publication are van Deursen, Mesbah, Cornelissen, Zaidman, Pinzger, and Guzzi.
- "Supporting Collaboration Awareness with Real-time Visualization of Development Activity" [109], published in the proceedings of the 14th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2010). The authors of this publication are Lanza, Hattori, and Guzzi.

1. This paper has been awarded a Best Paper award.

2 Developers' Teamwork

Teamwork in software engineering is time-consuming and problematic. In this chapter, we explore how to better support developers' teamwork, focusing on the software implementation phase happening in the integrated development environment (IDE), where developers spend most of their time.

Conducting a qualitative investigation, we learn that developers' teamwork needs mostly regard coordination, rather than concurrent work on the same (sub)task, and that developers have problems dealing with breaking changes made by peers on the same project, but they successfully deal with other scenarios considered problematic in literature. We subsequently derive implications and recommendations.[1]

1. This chapter contains the first part of the paper "Supporting Developers' Coordination in the IDE" [74], accepted for publication in the proceedings of the 18th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2015). The authors of this publication are Guzzi, Bacchelli, Riche, and van Deursen. This paper has been awarded a Best Paper award.

2.1 Overview

In the exploratory investigation presented in this chapter we aim at understanding how developers experience collaboration in teamwork and at discovering relevant and actionable problems developers face when working in team. To this end, we (1) conduct a brainstorming session with 8 industry experts who are working on the development of a popular IDE at a large US-based software company; (2) analyze the brainstorming outcome and surveyed related literature to derive three development scenarios to form the guideline for subsequent semi-structured interviews; (3) interview 11 industrial developers with various degrees of experience and seniority, from 9 different companies, about their teamwork experience using this guideline; and (4) further analyze the interview data (also using a *card sort* [164]) to make the most relevant findings emerge.

The results of our exploratory investigation show that developers work individually most of their time and, as a consequence, their needs in teamwork mostly regard managing dependencies between activities, rather than working together contemporarily on the same (sub)task. In addition, in the brainstorming the experts reiterate on the importance of shared information for teamwork, and underlined that it is the most valuable space for investigation. Finally, interviews, on the one hand, confirm that problematic development scenarios generated by lack of information (*i.e.,* imperfect information) are realistic; on the other hand, they also show that developers use effective mechanisms to avoid reaching those situations, or to quickly solve their consequences. One situation, though, is troublesome: Dealing with code changes, made by peer developers working on the same project, that generate errors. This situation emerges as one of the main causes of frustration in interviewees' experience of teamwork.

We derive implications from these findings and discuss recommendations for supporting coordination in the IDE.

2.2 Methodology

In this section, we define the research questions and outline the research method we pursued. In particular, we defined three research questions and divided our research method (illustrated in Figure 2.1) into three main steps: brainstorming, semi-structured interviews, and data analysis with card sorting.

2.2.1 Research Questions

Our investigation on how to better support teamwork within the IDE revolves around the following research questions:

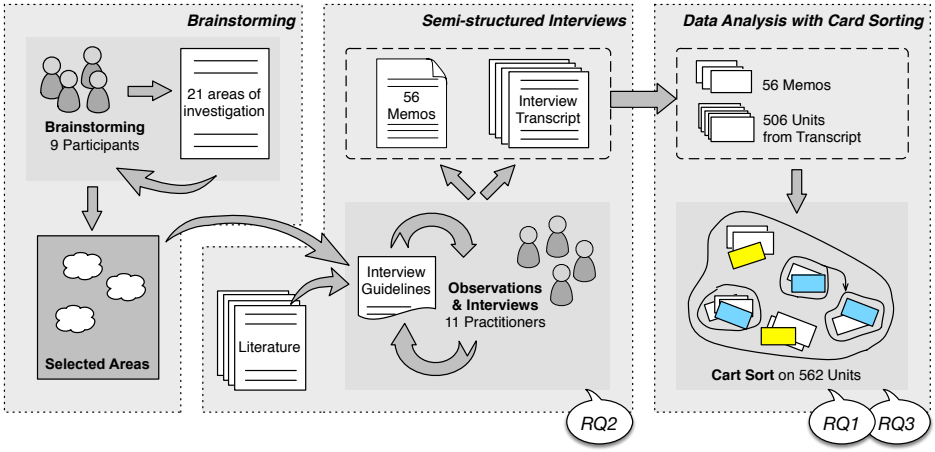


Figure 2.1: *The research method applied in the first phase*

[RQ1] What is collaboration in teamwork for developers?

[RQ2] How do developers face problematic development scenarios generated by imperfect information?

[RQ3] How do developers deal with receiving code changes?

Given the exploratory nature of our investigation, not all the research questions were known in advance. Answering the first research question was deemed necessary since the inception of this work (to reveal the most important aspects of teamwork for developers, to contextualize the answer of the subsequent questions, and to understand the leeway to introduce improvements in the IDE). The second and third questions emerged from the brainstorming and interviews.

2.2.2 Brainstorming

Empirical studies of software projects have shown that teamwork is at the basis of the most difficult and pervasive problems of software engineering, and that developers face a plethora of teamwork problems at different levels [37]. We scoped our initial focus by tapping in the collective knowledge of eight industry experts engaged in the design and implementation of development tools (including the main author of this research, as a researcher intern, and another author). To do so, the participants took part in a *brainstorming* session of 60 minutes. Brainstorming has traditionally been used to spur group creativity with the intention of generating concepts and ideas regarding a specific challenge [133]. After a brief training on the research context and goal, participants were divided into two groups. To kickstart the discussion, each group was invited to brainstorm on the challenges and opportunities revolving around two types of teamwork structures:

(1) one-to-one and (2) many-to-many. The two authors participating in the brainstorming were in different groups, their main role was to keep the focus of the discussion on the goals and to take notes on the resulting ideas.

After the brainstorm, we consolidated all the generated ideas into a list of 21 investigation areas. We emailed the list to the brainstorm participants, asking to sort the areas by the positive impact (based on their experience) that solving problems in those areas would have on teamwork. Our goal was to focus on the most relevant and promising area(s) to investigate.

The brainstorming led to the identification of the following areas for further investigation: *awareness* (*i.e.,* knowing other people's work), *work dependencies* (*i.e.,* knowing my dependencies and who depends on me), and *breaking changes* (*i.e.,* knowing who I might be impacting negatively with my changes and who impacts me). All these areas are centered on having certain pieces of information. In fact, for each of them, in the ideal scenario, all the information would be available to all developers, so that they would have all the relevant knowledge with which to make decisions (*i.e.,* situation of *perfect information*). However, in realistic teamwork environments not having all the information (*i.e.,* a situation of *imperfect information*) is the norm [88]. This is consistent with literature, which reported that developers in teams often have difficulties facing questions such as: "*What have my coworkers been doing?*" "*How have resources I depend on changed?*" "*What information was relevant to my task?*" "*What code caused this program state?*" [13, 59, 103, 162].

As a result of the brainstorming and its analysis, we proceed to investigate deeper the role of (im)perfect information in teamwork, through interviews with professional developers.

2.2.3 Semi-structured Interviews

After the brainstorming, we conducted interviews with professional developers to investigate their perception of collaboration in teamwork (**RQ1**), and how they deal with situations in which not having all the necessary information can be problematic (**RQ2**). To facilitate the discussion on the latter, we discussed it in terms of three concrete problematic scenarios generated by imperfect information. We derived the scenarios (detailed in Section 2.3.2) from teamwork situations commonly described as problematic in literature.

We conducted *semi-structured* interviews [113], a form of interviews that make use of an interview guide containing general groupings of topics (such as our scenarios) and questions, rather than a pre-determined exact set and order of questions. Semi-structured interviews are often used in an exploratory context, such as ours, to "*find out what is happening* [and] *to seek new insights*" [175]. We used the first research question and the scenarios as our guidelines.

We conducted 11 interviews, each one with a different professional software developer who was part of a team of at least four people. To increase the breadth of our analysis, we selected developers working in different companies and with varying experience levels (ranging from few months to more than 25 years) and time spent in the current teams

(ranging from few months to 10 years), in various combinations. For example, we interviewed a developer with 25 years of experience who worked in the same team for 10 years, and another with 21 years of experience who just joined a new team in a new company. Table 2.1 summarizes interviewees' characteristics.

Table 2.1: *Interviewed developers*

ID	Overall experience	In current team		Team Size
		Time	Role	
D1	7.5 years	4.5 months	dev	4
D2	10 years	6 years	dev lead	7
D3	2 months	2 months	junior dev	4
D4	1.5 years	1.5 years	sql dev	5
D5	20 years	10 years	senior dev	4
D6	25+ years	7 years	senior dev	15
D7	21 years	2 weeks	senior dev	10
D8	25+ years	10 years	senior dev	16
D9	1 year	1 year	dev	5
D10	15 years	5 years	dev lead	5
D11	20 years	6 years	senior dev	11

We conducted each 90-min interview on the phone, and transcribed the interviews for latter analysis. After each interview, we analyzed the transcript and split it into smaller *coherent units* (*i.e.,* blocks expressing a single concept), which we summarized by using either an interview quotation or an abstractive sentence. In addition, the interviewer kept notes (*i.e., memos*) of relevant and recurring observations in a document iteratively refined and updated.

Completed the interviews, we analyzed the resulting 56 memos, and we could answer our research question on how developers face problematic scenarios generated by imperfect information (**RQ2**). This answer opened a new question on how developers deal with receiving code changes (**RQ3**).

2.2.4 Data Analysis With Card Sort

A *card sort* is a technique employed in information architecture to create mental models and derive abstractions from input data [164]. We used this technique to answer our research question about how developers perceive collaboration in teamwork (**RQ1**) and the newly emerged **RQ3** about how developers deal with code changes. For each research question, we conducted a different card sort over the summarized 506 coherent units (of the transcripts) and the 56 memos. Each card sort was used to organize the units and memos into groups to abstract and describe developers' experience and perspective on the specific research question.

Each card sort had three steps: (1) *preparation* (select card sort participants and create the cards); (2) *execution* (sort cards into meaningful groups); and (3) *analysis* (form abstract hierarchies to deduce general categories).

Preparation: We created the cards from the 506 transcribed coherent units and the 56 memos, for a total of 562 cards. Each card included: the source (transcript or memo), the interviewee's name (if from the transcript), the unit/note content, and an ID for later reference.

Execution: The main author of this research analyzed the cards applying *open* (*i.e.*, without predefined groups, as they emerge and evolve during the sorting process) card sort. Categories were developed iteratively, and the sorting was redone multiple times, as the categories emerged, to strengthen the validity of the final result.

Analysis: After macro categories were discovered in the execution phase, the author who did the first card sorting and another author of this research re-analyzed the categorized cards to obtain a finer-grained categorization. Subsequently, we use *affinity diagramming* to organize the categories that emerged from the card sort. This tool is a process used to externalize and meaningfully cluster observations and insights from research, keeping findings grounded in data [116].

2.3 Results

In this section, we present the answers to our three research questions. We refer to information from the interviews using the [D*X*.*Y*] notation, the *X* represents the developer, and the (optional) *Y* the card (e.g., [D2.03] refers to card number 03 from the interview with developer D2).

2.3.1 RQ1: Teamwork from the developers' perspectives

According to the interview participants, collaboration in teamwork is defined by a wide spectrum of activities:

Communication – Collaboration is communication: As D8 said: "*It is all about communication. If you have good communication, you have good collaboration.*" Communication can be both one-to-one and one-to-many, can be formal or informal, and goes through different channels. Channels are "conventional," such as email and instant messaging (IM), but also domain specific, such as interactions via project management tools (*e.g.*, source code management systems and requirement tracking systems) and source code; as D4 explains: "[to communicate] *typically we use* [IM], *but we also have an internal wiki that we use.*" [D4.02, D5.100, D7.09, D8.(01,02)]

Helping each other by sharing information – As D9 said: "*Collaboration is just sharing information and ideas.*" In particular, according to interviewees, collaboration means

being proactive and sharing useful information to make it easier for others to complete their work (*e.g.,* "*make* [the acquisition of information] *as easy as possible on the other co-workers, so that they don't have to struggle*" [D7]). This aspect of collaboration involves voluntarily sending notifications (*e.g.,* FYI–*for your information*–messages) and starting discussions (*e.g.*, "*let's coordinate on this change I need to make*") [D7]. Resource sharing involves not only knowledge on the actual source code of the project, but also information from external resources, for example about new technologies or coding style; as D9 stated: "*We also like send each other things, like, style tips and like interesting articles about how other companies do things.*" [D2.02, D7.(02,03,06), D9.(01,04,05)]

Knowing what others know – According to the interviewees, collaboration also means to stay aware of the experts of the different parts of the system (*i.e.,* the domain experts) and to understand artifacts and code changes done by colleagues. D11 explains: "[collaboration] *is keeping track of what everybody is working on and being able to know how the different pieces are in place.*" According to interviewees, knowing what the others know has the aim both of preventing problems and of reacting faster when they occur. [D2.01, D4.03, D7.04–07, D11.47]

Working on the same goal, doing different things – Overall, developers see collaboration as working toward the same goal (*e.g.,* product releases), by acting in parallel on different parts of the same project (*e.g.,* working separately on different code artifacts): "*Collaboration is we are all working towards the common goal, but maybe working on different parts of it, but these components do interact*" [D7]; "[Collaborating meant] *we divided up the work* [...], *we went off these directions, and as things started to merge together, we go on* [merging] *on a case by case base*" [D3]. [D1.02, D3.01, D4.01, D7.01, D9.(02,06)]

2.3.2 RQ2: Dealing with imperfect information in teamwork

Imperfect information emerged from the brainstorming and its analysis as the issue creating most difficult situations in teamwork. Our second research question seeks to understand how developers deal with these situations, in order to define actionable areas for supporting teamwork.

To investigate how developers deal with imperfect information, we outlined three concrete teamwork scenarios in which the existence of imperfect information can generate problems. The scenarios were derived from teamwork situations commonly described as problematic in literature.

Inefficient Task Assignment

Scenario. One developer is assigned to a task, while another is already working on a similar or related task. This introduces inefficiencies in the team.

Related literature. Software engineering researchers have recognized task partition and allocation as important endeavors in the context of teamwork [102, 105]. If dependencies and common traits among tasks are not correctly handled, developers can reduce their efficiency and generate unexpected conflicts [85]. Literature suggests different techniques, with varying results, for efficient task assignment (*e.g.,* [30, 52]). In particular, the assignment of bug fixes (or new features to implement), from a repository of issues or requests to the most appropriate developers, is one of the main instances of task assignment investigated by researchers [1]. Researchers reported that bug triaging is a time consuming task, which requires non-trivial information about the system, and that often leads to erroneous choices [96]. A number of techniques has been devised to tackle the triaging problem, *e.g.,* [118, 94].

Interviews' outcome. Although considered realistic, this scenario was not seen as a core issue by interviewed developers. In fact, the task assignment processes of teams are in general effective to prevent the problematic scenario to take place. In some teams supervising figures (*e.g.,* managers) do the task assignment (*e.g.,* "[a new task] *goes to a manager, who decides whom to assign*" [D8], and "*the boss will tell you about a task* [to do]" [D9]); in the other teams, tasks are divided during team meetings, with various degrees of developers' interaction (*e.g.,* "*we are using daily SCRUM meetings*" [D1], and "*we break up the code, and if the* [task] *is in your code, it's yours*" [D5]).

Simultaneous Conflicting Changes

Scenario. Developers find themselves in a situation where there is a merge conflict (*i.e.,* when different people are touching the code at the same time).

Related literature. A recent significant effort in the software engineering research community is devoted to detect concurrent modifications to software artifacts (*e.g.,* [82, 83, 141, 151]). In fact, developers making inconsistent changes to the same part of the code can cause merge conflicts when changes are committed to the code repository, which leads to wasted developers' efforts and project delays [24, 99, 152]. Grinter conducted one of the first field studies that investigated developers' coordination strategies [67]. She observed that it is sometimes difficult for developers (even for expert ones) to merge conflicting code without communicating with the other person who worked on a module. Later, de Souza *et al.*, in an observation of the coordination practices of a software team at NASA, observed that developers in some cases think individually trying to avoid merging, while in others they think collectively by holding check-ins and explaining their changes to their team members [43].

Interviews' outcome. Our participants reported only rarely encountering a situation where more than one person was working on the same file at the same time ("*we don't run into those situations a lot*" [D2]). Most of our participants' teams were organized to make developers work on different parts of the system, with a key person in charge of coordinating development to prevent those issues, typically a lead developer or an architect. Some participants also used technical solutions to avoid concurrent edits (*e.g.,* "*We*

put a lock on [the file], *so it does not get edited* [by others]" [D1]). When a merge conflict happens, our participants reported resolving it quickly and easily (*e.g.,* "*The best and quickest solution you have is to undo, we roll back and* [fix it]." [D1]; "*typically, it is solved really quick*" [D2]), and often using merging tools (*e.g.,* "*we don't have to do much manually*" [D8]). Although automatic merging was used, our participants also explained that they manually checked each conflict, revealing that it is not entirely trusted.

Breaking changes

Scenario. A developer/team changed some parts of source code. This introduces a change that breaks the code of another developer/team.

Related literature. Researchers consider breaking changes problematic, not only for developers who receive the change and have to understand and adapt their code, but also for developers who are making a change that might break the functionalities of many clients [2]. Literature presents investigations (*e.g.,* [44]) on the effect of dependencies that break other people's work, and proposed methods to address the problems they cause, at the scale of both single system (*e.g.,* [19, 177]) and ecosystems (*e.g.,* [76, 147]).

Interviews' outcome. The reaction of the participants were different according to the *origin* of the breaking changes. When the origin of the breaking change is *external* to the company or, more in general, when developers feel they have neither space for intervening nor displaying their disappointment to 'the breaker', they accept the breaking changes with no strong negative emotions but rather acceptance. This happens even when resolving the issue might take a vast amount of time (*e.g.,* "*more than a year*" [D2]) or have a high economical cost (*e.g.,* "*this* [break] *was costing the company many thousands of dollars per minute.*" [D7]). However, when the origin of the breaking change is *internal* to the company/team, it causes strong negative emotions (*e.g.,* frustration). These emotions seem to be due to the long time spent by the developer in finding the cause of the issue, which is then resolved relatively quickly (*e.g.,* "*I spend a couple of hours to find out the error* [...] *fixed in 5 minutes.*" [D3] and "*I spent a day fixing the problem I spent three days finding.*" [D8]). Breaking changes generating syntactical errors were not considered an issue, because they could easily be spotted and fixed. While time is not an issue for external changes, it is for internal ones. This seems to be a direct consequence of the lack of coordination effort that was expected from the breaker [D1]. Some developers call for an effective way to deal with receiving breaking changes, when the origin of the break is internal to the team. For example, some would prefer stricter rules to avoid internal breaking changes: "*people breaking other's people code* [...], *I'd like to see management being more rigorous about it*" [D8].

2.3.3 RQ3: Receiving a code change

Managing internal breaking changes is the most problematic scenario emerged from **RQ2**; our third research question seeks to analyze how developers deal with changes made by

peer developers working on the same project. In the following we answer this question by describing how interviewed developers generally deal with receiving code changes.

When developers are notified of a change in the codebase (*e.g.,* via automatic emails from versioning systems), they decide to investigate it mostly to discover whether it has an impact on their own code. Less frequently, their aim is to update/build their own knowledge of the codebase with information such as code expertise and code ownership. When this process leads to discovering that the changed code has an impact on their own part, developers proceed to analyze the change impact. In the few cases in which a change has been made to their own code, developers investigate it.

When developers discover an unexpected error (or an alteration in the behavior) caused by a change made by someone else, their most prominent complaint regards the lack of coordination that they feel should have accompanied the change (*e.g.,* they would have expected an "heads up" email). However, in the case of clear syntactic errors (*e.g.,* compilation errors, or runtime errors generating a stack trace), developers do not feel as much frustration as in the case of semantic errors (*e.g.,* caused by a library that changes some sorting order, therefore impacting the dependent code) or unclear alteration in the behavior. In fact, semantic errors require developers to perform a deeper and more time-consuming analysis to understand their cause [D3.(47,49), D8.(28,29,36)].

Once they find the cause, then they can proceed with measuring the impact on their code by, for example, measuring how many files or lines of code needs to be changed (as D8 explained: "*I measure the impact of a change* [looking at] *how many files/lines it affects. A few? Hundreds?*"). Usually, the developers receiving the breaking change are the ones automatically responsible of adapting and fixing their own code. However, when a change has a deep impact on the codebase, which requires more information about the change (*e.g.,* the rationale of the change) and the codebase, developers usually want to get in touch with the author.

When the change introduces a defect, developers receiving it decide whether filing an issue report against the change to the change author (*e.g.,* D5 explained: "*if the bug is in your code, it's your bug to fix.* [...] *I send a bug request.*") [D3.(06,09,31), D5.06], or, occasionally, if the fix takes little time, directly changing the faulty code to fix it (D3 said: "*If it's small I just fix it and notify the author*") [D3.59, D5.107, D8.38, D9.60]. The time that gets spent fixing the problem in the code does not seem to bring frustration *per se* (*e.g.,* "*Diagnosing is almost always harder than to fix it. With the majority of bugs, once you know where the problem is, it's easy to fix*" [D8.29]). The rationale to fix it directly is that the developer already has all the necessary information to fix it, which would require time to share with the author of the faulty code.

Some developers mentioned that the lack of testing contributes to faulty changes being committed to the repository (*e.g.,* "*we are really bad at testing* [...]*, you pull and you get a file you try to run and it fails*" [D9], "*If I'd tested it better, I wouldn't have put [this code] in the build*" [D5]). Nevertheless interviewed developers also warned that running all the tests for each change would be too expensive ("*all tests, to run them all, it would take 3 weeks.*

Unfeasible to take 3 weeks for each check in" [D6]), that testing working on a setup might unreliable on a different one ("*we test and it's all good, but then they test on their end and it might break* [...]. *It's something to do with customizing.*" [D2]), and that many semantic changes could not be detected by tests ("*even if there are tests that check* [most] *things, you'd still end up with edge cases.* [...] *You still need to see that you break, and then react, and then fix it*" [D6]).

2.4 Interpretation, Implications and Recommendations

In this section, we explore the meaning of the data gathered from our three research questions and discuss its implications and ways in which collaboration in teamwork can be improved.

2.4.1 Teamwork Collaboration is Coordination

The terminology used in many disciplines [115] defines coordination as "*managing dependencies between activities,*" and collaboration as "*peers working together.*" In this light, what participants consider as collaboration in teamwork is mostly *coordination*, needed to organize the individual work toward achieving a shared goal.

By analyzing the data from the interviews, coordination emerged—after individual work—as the dominant level of interaction when working in team, rather than collaboration. In particular, the interviewed developers interact as follows:

1. Developers spend most of the time doing individual work;
2. Most of developers' interaction is to coordinate (*e.g.,* through daily stand-ups);
3. Collaboration happens infrequently and on a need basis (*e.g.,* with (bi-)weekly sprint meetings).
4. Most of the time, the intention of collaboration is coordination, which leads to individual work eventually.

Both collaboration and coordination are tied to the level of interaction among the developers working together.

By abstracting the explanations of interviewees, we model developers' interaction in three levels (from lowest to highest degree of interaction): individual work, coordination, and collaboration. Individual work corresponds to no interaction (*e.g.,* a developer working on her own), while collaboration means developers working together at the same time on the same (sub)task (*e.g.,* pair programming). Coordination is an intermediate level of interaction, where developers interact but are not working together on the same (sub)task (*e.g.,* during task assignment).

An activity is a working state that can be placed as a point on the corresponding level of interaction. In a set of activities done to accomplish a certain subtask (*i.e.,* 'working

situation'), there can be steps between different levels of interaction, for example, steps from individual work to coordination (*e.g.,* "[when a file is locked] *we just* [ask]*: 'hey what are you working on? And then when you think I can do it?' to the author.*" [D2.10]), and from coordination to collaboration (*e.g.,* "*sometimes we* [...] *get together and talk about* [similar things]*, then realize how we can do these things together and do them together*" [D11.45]). Figure 2.2 depicts our model of developers' interactions.

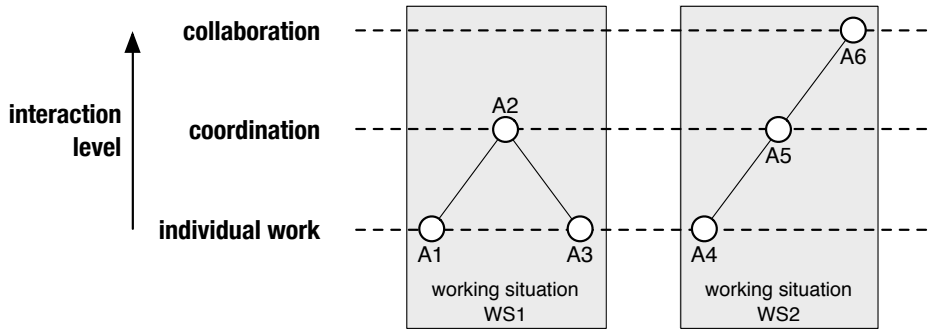


Figure 2.2: *Model of developers' interactions in working situations*

Figure 2.2 shows two working situations: In the first (WS1), a developer doing individual work asks another developer to make a change in their code (*e.g.,* "*I asked one of the guys: '*[...] *I need a method that would return* [a special object]*, can you write* [it] *for me?' He was able to write* [it] *and knew exactly where to go*" [D3.(09,15)]). This requires going from individual work (A1) to coordination (A2) when asking to make a change to the other, and back to individual work (A3) when they reach an agreement, without reaching a state of collaboration. In the second situation (WS2), two developers decide to work together on a subtask. This requires moving from individual work (A4) to coordination (A5) when they decide, then to collaboration (A6) for the actual work.

The steps between the different levels of interaction in the model are not discrete: Intermediate interaction levels can be reached. For example, while the activity of task assignment can generally be placed on the coordination level, when the task assignment is *discussed together in a meeting* it lays on a level between coordination and collaboration.

Implications and Recommendations

Our participants report that most of their time is spent in doing individual work, while, unexpectedly, they report spending very little time on working together on the same subtask (*i.e.,* collaborating). A direct consequence is that interactions revolving around coordination are a more actionable area, with better research opportunities and with greater potential impact, than areas considering purely the broader collaboration aspects. For

example, better support for communication would have more relevance than tools for concurrent real-time coding.

In addition, techniques for supporting information sharing among developers should take into account that developers spend most of their time doing individual work. Considering that most of this individual work is spent in the development environment (the IDE) [110], tools that support coordination within the IDE have the potential, at least, to be used more frequently and to not require developers to drastically change their working habits.

2.4.2 The Role of Information

Information is what developers need to make the step from one level of interaction to a higher one (Figure 2.3). Developers can already have this information, or they can gather it, for example, through communication or by understanding code changes done by colleagues.

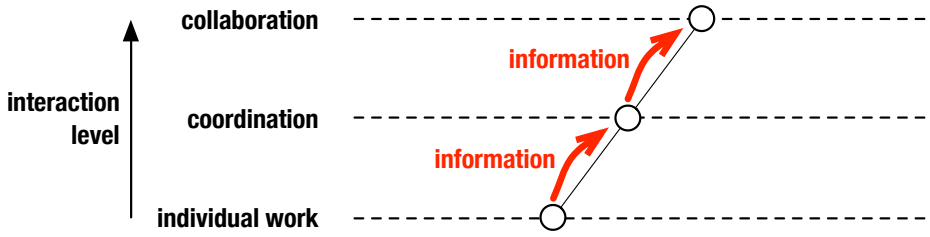


Figure 2.3: *The role of information in developers' interaction*

In an ideal setting, developers would work in a situation of perfect information. This would allow seamless transitions from one level of interaction to the higher one. Nevertheless, developers do not have all the information necessary to complete a subtask, thus making the step harder to do.

The importance of information sharing for teamwork, which also emerged from our brainstorm (Section 2.2.2) as the most valuable space for investigation for the aim of our study, has been studied by researchers over the years from different angles. For example, Cataldo *et al.* introduced the notion of *socio-technical congruence* and provided evidence that when developers share information when making changes in dependent software components, they complete tasks in less time [29]. Other researchers similarly showed that missing information correlates with coordination problems and project failures [28, 150, 108]. Ko *et al.* analyzed the information needs of developers in collocated teams and found they

have difficulties in answering questions related to information about colleagues' work and software component dependencies [103].

From our interviews, we observed that developers know how to deal with the investigated scenarios involving imperfect information, except when they receive an internal breaking change. This is connected to how easy it is to access the information they need. Analyzing the ways developers/teams successfully deal with a condition of imperfect information, we see that the solutions to the problematic scenarios require information to be shared in two ways: (1) via direct communication (*e.g.,* during a meeting), and (2) by making it visible (*e.g.,* in a tool).

Table 2.2: *Information in investigated scenarios*

Scenario	Needed Information	
	Communicated	Visible
Task assignment	✓	✓
Simultaneous changes	✗	✓
Breaking changes	✗	✗

Table 2.2 shows that for manageable scenarios, the needed information is communicated or visible. In the case of task assignment, inefficiencies are avoided by centralizing the task assignment to the team leader, who has all the information "visible" in mind, or by conducting group meetings in which the information is communicated. Other researchers report evidence of this behavior: Begel *et al.* described that industrial program managers and developers have regular team meetings for effectively prioritizing bugs and to coordinate component completion schedules [14]; and Guzzi *et al.* reported that when open source software developers meet in person at conferences manage to advance the coordination of their projects better [73]. In the case of simultaneous changes that were not avoided with the team policies (*i.e.,* through modularization and technical locks), the information necessary to solve the merge conflict is immediately visible through the merge tool. de Souza and Redmiles similarly reported that parallel development scenarios they analyzed were not problematic because configuration management tools can handle most of the situations involving merges [44]. In the case of breaking changes, we see that the needed information is neither communicated in time nor easily accessible/visible. As a result, developers spend long time in finding the information they need to coordinate. A number of previous studies reported that breaking changes are due to missing information and lead to significant loss of time (*e.g.,* [147]).

Implications and Recommendations

Our analysis showed that the efforts spent in gaining information, which developers are missing due to lack of coordination within the team, are a source of negative emotions.

This underlines the importance that information should be shared (*e.g.,* communicated) and accessible (*e.g.,* visible via a tool) when working in a team.

Researchers proposed a number of tools (*e.g.,* Palantìr [151] and FASTDash [16]) to detect merge conflicts and tested them in laboratory settings with seeded conflicts. These tools helped developers to spend less time to resolve conflicts and encouraged communication. An interesting venue for future research is to verify the overall impact of these tools on teams whose structure reflects the software architecture, where participants reported this problem to be less frequent.

In addition, in most of our investigated scenarios, we observed that—unexpectedly—developers already had means to deal with the missing information, or did not consider these scenarios as issues, thus making them less worth of further investigation. On this note, the results of the study by deSouza and Redmiles put in evidence the significant differences that two unrelated companies have when they deal with the management of dependencies and the underlying information sharing [44]. This suggests that what is considered a critical issue for a company/project could be not important for another. As a consequence, it is important, when investigating potential problems generated by lack of information, to first study whether and how the target developers already employ any method to supply this missing information.

What emerged as a fertile area of research is investigating ways to support developers dealing with breaking changes. In fact, developers do not currently have an easy way to find and access the necessary information. Particularly important is the case of internal breaking changes because in this situation developers feels the strongest negative emotions, in view of the fact that the needed information could have been made available by their colleagues.

2.4.3 Code Changes and Dependencies

As de Souza and Redmiles explained: "it is not possible to study changes in software systems without studying dependencies" [44]. In this light, our analysis of coordination and receiving changes is related to the work by Begel *et al.* [14] and by de Souza and Redmiles [44].

Begel *et al.* investigated coordination and dependencies in large-scale software teams at Microsoft. They conducted a survey of Microsoft developers, testers, and program managers to see how these coordinate on dependencies (*e.g.,* tasks) within the same team and among different teams. The study reported that most Microsoft developers (63%) minimize code dependencies to mitigate problems with dependencies. This is similar to our interviewees who use software modularity to avoid inefficient task assignment or merge conflicts. Similarly to our findings, Begel *et al.* also reported that lack of communication causes most coordination problems, and that developers keep track of items they depend on by using emails. Besides these similarities, the different context let emerge interesting differences, particularly in the dichotomy between internal and external dependencies

and changes. Begel *et al.* found that internal dependencies are managed by "send[ing] an email and pay[ing] a personal visit to the person blocking their work," [14] and surveyed developers do not report any negative emotion. Our findings underlined that, in the case of internal breaking changes, the process preceding the communication with the person blocking the work (*i.e.,* finding the source of the problem) is cause of dissatisfaction and frustration, because the expected communication did not take place. This situation may be problematic for the productivity of the team. Moreover, the two studies present different definitions of *external* dependencies and breaking changes: (1) According to Begel *et al.*, dependencies are 'external' if in different teams within the same company, with which it is possible to communicate personally; (2) according to our findings, dependencies are 'external' if in different companies, with which it is extremely difficult to communicate. In the former case, Begel *et al.* reported that developers have to maintain personal communication with external teams to remain updated of changes, and the existence of unexpected changes from external teams generates anxiety. In the latter case, our interviewed developers did not report anxiety (even though unexpected changes happen and lead to loss of time), rather acceptance of the situation as part of the natural business model of the industry.

In their work, de Souza and Redmiles investigated the strategies software developers use to handle the effect of software dependencies and changes in two industrial software development teams [44]. The two teams deal with *internal* dependencies according to our definition. One team (MVP) allows parallel development and the modularity of the system is low, the other team (MCW) focuses on modularity by using a reference architecture. Our interviewed developers have complains similar to those in the MCW team, and these teams have strikingly similar practices: In both studies these teams avoid inefficient task assignment with modularity, their developers have problems identifying their impact network (they do not know who is consuming their code or whether changes can modify the component they depend on) and are only interested in changes in particular parts of the architecture that impact them. Moreover, both developers in MCW and our study have expectation that major changes are accompanied by notifications about their implications, and their are also worried about receiving too many notifications. On the other hand, the MVP practices correspond to the ideal scenario for our interviewed developers: Emails are sent to update about changes, everybody reads the notification emails just to remain updated, management urge developers to notify about breaking changes and the latter even suggest courses of action to be taken to minimize the impact. As a result, despite the parallel development, coordination in MVP seems smoother than in our cases. One important feature of MVP, mentioned by de Souza and Redmiles, is that most developers have worked on the project for at least two years, and their experience could also be the cause of the difference with MWC, which is a younger project. Our results, though, do not seem to corroborate this hypothesis, since interviewed developers reported similar issues regardless of age of project and their experience. Our additional analysis of code changes looks at coordination from a low level perspective: Through it, we additionally found that most of the information developers need to coordinate would be available, but it is not easily accessible in their working environments.

Implications and Recommendations

Our study confirms that lack of coordination leads to late discovery of unexpected errors or behaviors in the code, followed by a time-consuming analysis to find the code changes that are the source of the error/behavior. This calls for better support for coordination when developers make and receive changes (*e.g.,* a tool that informs developers automatically whether a change they are making has an impact on someone else's code), and for when they need to investigate a change to determine its impact. Although research has been conducted on these topics (impact analysis and support for change understanding), they are still open issues and research prototypes have not yet reached widespread usage in the IDE. Our findings, in line with previous studies, underlines the substantial practical relevance of further research in these areas.

The differences between coordination practices between our interviewees' teams and the MVP team described by de Souza and Redmiles [44] are an interesting venue for future research. In fact, compelling is the hypothesis that the modularity adopted by our interviewees' teams and MWC could create asymmetries in engineers' perceptions of dependencies [68], thus being at the basis of the differences and generating the reported coordination issues.

By investigating how developers currently handle received code changes in the IDE, we realized that they do many tasks manually, and spend a lot of effort to collect and remember change information. The data that would help developers in their tasks is available (*e.g.,* historical data recorded by the versioning systems), but not easily accessible. This situation implies that better support for integrating change information in the IDE is needed, and that it would have impact on software development and coordination.

2.5 Concluding Remarks

In our study we explored how to support developers' collaboration in teamwork. We focused on teamwork in the software implementation phase, which takes place in the IDE, and we conducted a qualitative investigation to uncover actionable areas for improvement. We identified internal breaking changes as one of the most important areas for improvement, because current IDE support for receiving changes is not optimal.

This chapter makes the following main contributions:

1. A qualitative analysis indicating that teamwork needs mostly regard coordination, that developers are able to face scenarios considered problematic in literature, and that dealing with breaking changes is hard, but it only generates frustration if the breaker is internal to the project.
2. Recommendations on how to improve collaboration in teamwork in the software implementation phase, such as to focus on interactions revolving around coordination rather than on collaboration on the same (sub)task.

3 Communication in OSS Development Mailing Lists

Open source software development teams use electronic means, such as emails, instant messaging, or forums, to conduct open and public discussions. Researchers investigated mailing lists considering them as a hub for project communication. Prior work focused on specific aspects of emails, for example the handling of patches, traceability concerns, or social networks. This led to insights pertaining to the investigated aspects, but not to a comprehensive view of how developers communicate. Our objective is to increase the understanding of development mailing lists communication.

We quantitatively and qualitatively analyze a sample of 506 email threads from the development mailing list of a major open source software project, Lucene. Our investigation reveals that implementation details are discussed only in about 35% of the threads, and that a range of other topics is discussed. Moreover, core developers participate in less than 75% of the threads. We observe that the development mailing list is not the main player in open source software project communication, as it also includes other channels such as the issue repository.[1]

1. This chapter contains the paper "Communication in OSS Development Mailing Lists" [73], published in the proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013). The authors of this publication are Guzzi, Bacchelli, Pinzger, Lanza, and van Deursen.

3.1 Overview

Open source software (OSS) development teams use electronic means, such as emails, instant messaging, or forums, to communicate. Conversations in OSS settings are typically conducted in an open public manner and are stored for later reference [17], thus OSS communication repositories offer a rich source of historical information, which can be used, for example, to observe software processes [158], to understand software developers communication dynamics [155], and to improve development practices [159].

Among the many OSS communication repositories, researchers focused on mailing lists (*e.g.,* [121, 18, 145]), as they have been—historically—the communication hub at the inception of successful OSS systems, such as Linux [144] and the Apache server [121].

Nevertheless, we are lacking a clear, updated, and well-rounded picture of the communication taking place in OSS development mailing lists. We only have either abstract and outdated knowledge (*e.g.,* obtained as a side effect of the analysis of OSS projects such as Linux [144]), which does not take into account the recent shift of interest to new social platforms (*e.g.,* GitHub, Jira), or a very specialized understanding (*e.g.,* regarding specific information, such as the code review process [145]), which does not take into account all the information we can distill from development emails.

Our goal is to increase our understanding of development mailing lists communication: What do participants talk about? How much do they discuss each topic? What is the role of the development mailing lists for OSS project communication? Answering these questions can provide insights for future research on mining developers' communication and for building tools to help project teams communicate effectively.

To answer these questions, we conduct an in-depth analysis of the communication taking place in the development mailing list of one major OSS software system, *i.e.,* the Apache Lucene project. We set up our study as an exploratory investigation. We start without hypotheses regarding the content of the development mailing list, with the aim of discovering the topics of communication, the prominency of implementation details, the position of developers, and the role of the development mailing list as communication channel. To that end, we manually inspect and classify 506 email threads comprising over 2,400 messages, we manually resolve the aliasing among more than 310 email addresses, and focus on gaining a holistic view on the information exchanged in the mailing list.

Our results show that, although the declared intent of *development* mailing list communication is to discuss project internals and code changes/additions, only 35% of the email threads regard the implementation of code artifacts. Instead, development mailing list communication also covers a number of other topics, such as social norms and infrastructure. Also, project developers participate in less than 75% of the overall threads and they start only half of the discussions. Finally, the development mailing list is not the sole player in OSS project communication: It is complemented by other channels (*e.g.,* issue repository) from which it is disconnected.

In this chapter, we make the following contributions:

- A coding system that is reusable for analysis of developer communication in general, and mailing lists in particular (Section 3.4).
- An assessment of relative frequency of topics in developer mailing lists (Section 3.5).
- An assessment of relative participation of developers in developer mailing lists (Section 3.6).
- A qualitative evaluation of the role of development mailing list for project communication (Section 3.7).

Based on our findings, we analyze and discuss the implications for researchers and practitioners (Section 3.10).

3.2 Related Work

By analyzing OSS development mailing lists, researchers provided insight in social aspects of software development. For example, researchers exploited email *metadata* (*e.g.,* author, date, and time) to conduct quantitative social analyses: Bird *et al.* proposed techniques to mine email social networks [17], and investigated social interactions in OSS projects [18]; Ogawa *et al.* visualized social interaction among participants in OSS projects [129]; and Shihab *et al.* showed that mailing list activity is related to source code activity [159]. Researchers also quantitatively analyzed the *text* of emails: Pattison *et al.* studied the frequency with which terms of software entities are mentioned in emails, and correlating it with the number of system changes [136]; Baysal and Malton searched for a correlation between discussions and software releases [10]; and Bacchelli *et al.* analyzed the correlation between email discussions and software defects [4].

Most of the aforementioned work is quantitative and based on the premise that development mailing list communication mostly regards the implementation of source code artifacts. This assumption derives from the knowledge about OSS systems provided by seminal literature such as *The Cathedral and The Bazaar* [144]. Few studies analyzed the *content* of OSS mailing list communications and mostly focused on specific traits of the communication. Gutwin *et al.* read mailing list archives to study group awareness in distributed development [70]. Rigby *et al.* analyzed mailing lists to study the OSS code reviewing process (*e.g.,* [145]). Mockus *et al.* studied the Apache Server development process finding that the mailing list play a central role for communication, coordination, and awareness [121]. We want to obtain a comprehensive knowledge of communication in development mailing lists of OSS projects.

Our work is also related to data quality: By knowing what data is available in mailing list repositories, we can devise better techniques for extracting relevant, unbiased, and comprehensible information. In this vein, researchers have studied bug repositories [179] and code repositories [100] to understand what information is more relevant. They also analyzed the impact of data quality on mining approaches and analyses (*e.g.,* [126]). In

the context of mailing list data, Bettenburg *et al.* showed the risks of using email data without a proper cleaning pre-processing phase [15].

3.3 Methodology

To explore and understand the communication taking place in development mailing lists, we performed an in-depth analysis of the development mailing list of Apache Lucene, an OSS information retrieval framework and API.

We chose Lucene for the following reasons: (1) Lucene is a mature project with a large user base and an established community of developers; it is a top-level Apache project since 2005, and it has been broadly recognized for its role in the implementation of search engines. (2) It was started in 1999 by a single developer, who initially guided it as a "benevolent dictator"; in 2001, Lucene joined the Apache Software Foundation and became a *foundation*, with a well-organized, hierarchical governance structure and formalized policies.[2] (3) The previous work describing the communication occurring in the development mailing list of OSS projects (*e.g.*, [144, 107]) dates back to the early 2000s, it is high-level and focuses on Linux, which is more of an exception than the rule in OSS projects [144]. Lucene's organizational structure sets it apart from the benevolent dictatorship of Linux; by choosing Lucene we aim at having an updated knowledge of contemporary developers' communication in the development mailing list in a more common OSS setting. (4) Lucene has a *publicly* archived development mailing list with a *declared* intent: It is "*where participating developers of the Java Lucene project meet and discuss issues concerning Lucene* [...] *internals, code changes/additions, etc.*"[3]

3.3.1 Research Questions

To understand the current usage of an OSS development mailing list and how OSS developers communicate through this channel, we conduct our investigation to answer the following four research questions:

[RQ1] What topics are development mailing list participants talking about?

[RQ2] How often do participants talk about each topic? How prominent are implementation details?

[RQ3] Is the development mailing list just for developers? What do developers focus on?

[RQ4] What is the role of the development mailing lists for the communication in the OSS at large?

2. http://www.apache.org/foundation/
3. http://lucene.apache.org/core/discussion.html

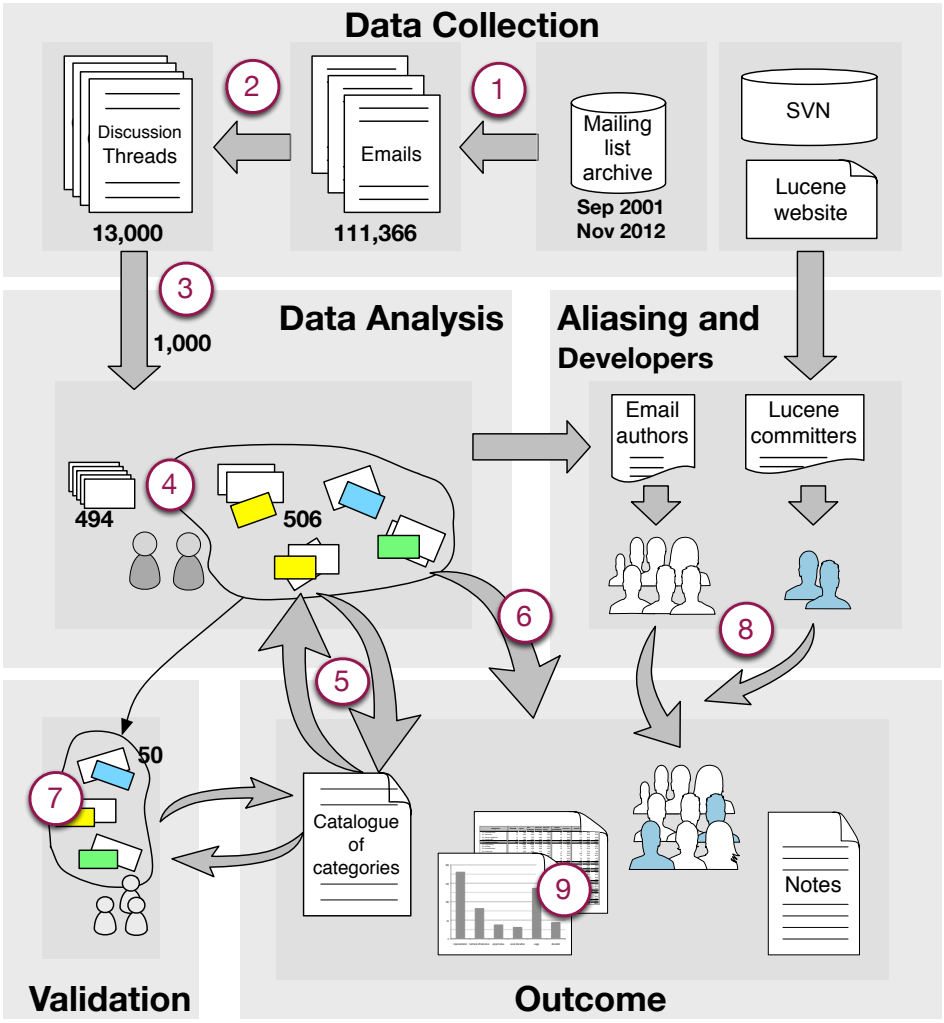


Figure 3.1: *The mixed approach research method applied.*

3.3.2 Research Method

We followed the approach depicted in Figure 3.1: We modeled all the mailing list emails (Point 1), reconstructed threads of discussion and removed auto-generated ones (Point 2), and randomly extracted 1,000 threads (Point 3). Using open card sort [164] (see Section 3.3.4), the first two authors of this research manually analyzed together the threads and extracted categories of discussion (Point 4). To ensure the integrity of the extracted categories, they sorted threads several times and iteratively refined the catalogue (Point 5). During the cart sort, they took notes about the mailing list, its role, and the communication occurring in it (Point 6). We validated the resulting catalogue of categories using closed card sort (Point 7). We complemented the automatically collected email data by resolving aliases and by adding information about which participants were developers (Point 8). Finally, we conducted a statistical analysis on the categorized threads (Point 9).

3.3.3 Data Collection

In our previous work, we presented Miler, a toolset to explore email data [5]. It crawls the website of MarkMail[4] an online service for searching mailing lists. MarkMail has two drawbacks: It obscures email addresses for privacy reasons, and it does not always reconstruct email threads. To correctly recognize participants, their roles, and the discussions threads, we extended Miler to collect, extract, and model data from a more complete source than MarkMail: Mbox files. This implies challenges also mentioned by Bettenburg *et al.* [15].

Extracting messages: We wrote an Mbox importer tool in Python to download and model emails. Although the Python library `mailbox`[5] gives reliable support for loading the different messages from Mbox files, we had to write an algorithm to automatically correct wrong date formats.

Reconstructing threads: An email *discussion thread* is a set of messages that are logically related, *i.e.,* replies in the same chain of emails. To reconstruct discussion threads, we use two complementary heuristics: (1) Whenever possible, we consider the 'message-ID' (a globally unique identifier for emails) and 'in-reply-to' (used to specify the 'message-ID' of the email that it is replying to) fields to reconstruct threads. (2) Otherwise, we consider email subjects. By manually inspecting the Lucene mailing list, we found that participants are conservative in keeping the subject consistent with the discussion: When a thread changes topic, participants accordingly modify the subject of the subsequent emails. Thus, if two emails have the same subject, or two slight variations of it (*e.g.,* they are prefixed by 'Re:'), we place them on the same thread, using the timestamp for sorting.

Removing automatically generated emails: Many OSS projects forward a number of automatically generated emails to development mailing list, for example, from the

4. `http://markmail.org`
5. `http://docs.python.org/2/library/mailbox.html`

versioning or the issue tracking systems. For the purpose of our research, aimed at understanding what *participants* talk about in a mailing list, we filter out these automatically generated emails, unless they are answered by a person. Although this filter has to be customized to the mailing lists under analysis, we used an approach that can be adapted to other lists. It focuses on the quantity and the thread subject. In fact, automatically generated emails often outnumber those manually generated and have a well defined subject pattern. We aggregated threads with a subject starting with the same 10 characters and manually analyzed their distribution. This approach found almost all generated emails.

3.3.4 Card Sort

To group the email threads we used *card sort*, a technique used in information architecture to create mental models and derive taxonomies from input data [164]. We used it to organize the threads into groups to abstract and describe mailing list communication. A card sort has 3 steps: (1) *preparation* (select card sort participants and create the cards); (2) *execution* (sort cards into meaningful groups); and (3) *analysis* (form abstract hierarchies to deduce general categories).

Preparation: We created all cards from the sample resulted from the data collection. Each card (exemplified in Figure 3.2) represents a thread and includes: (1) number of emails, (2) subject, (3) duration, with timestamp of the first and last emails, (4) the first 15 lines (removing white lines) in the body of the initial email, (5) email addresses of the participants involved, and (6) an univocal `id` for later reference.

Execution phase: The first two authors of this research analyzed the cards applying *open* (*i.e.,* without predefined groups, as they emerge and evolve during the sorting process) card sort, adapting the guidelines for the analysis of qualitative data with grounded theory [64]: They avoided information related to Lucene (*e.g.,* its website) and the literature closely related to mailing list communication, as this could have sensitized them to look for concepts related to existing theory, thus hindering innovation in organizing the threads. They often interrupted the card sorting to memo an idea or concept potentially useful for later analysis (see Section 3.7). When necessary they consulted the entire thread online. Since the rigor of the card sorting method is in its analysis [116], instead of working separately on different cards, and checking the consistency of the sorting and merging the cards in a later phase, they used *pair-sorting*. This requires significantly more time, but it brings more value to the analysis as they discussed discrepancies in their thoughts for each card during the card sorting itself.

Analysis phase: To ensure the integrity of the emerging categories, the first two authors did a second pass on all the analyzed cards, starting from small groups that could not be included in any larger group, and re-categorizing these cards by redefining some categories. Subsequently, they analyzed the remaining cards to completely describe the catalogue of thread categories (see Section 3.4).

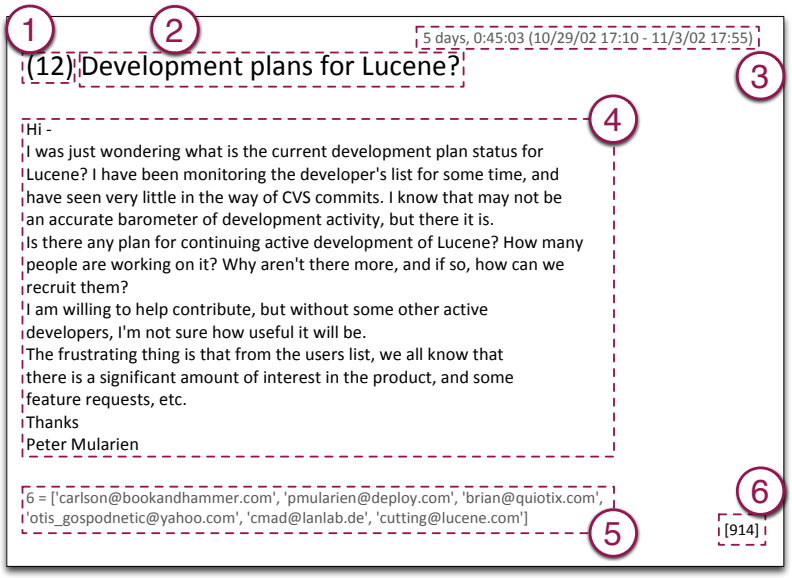


Figure 3.2: *Card Sort: Example Card*

We conducted a validation to verify whether the catalogue was written in a clear and understandable way that was capturing all the facets of each category (see Section 3.9).

3.3.5 Aliasing and Identification of Developers

Resolving multiple identities (aliases) is fundamental to prepare mailing list data for the statistical analysis of the participants [18]. Although a number of approaches to solve aliasing have been proposed, including the one presented in Chapter 4 (Section 4.4.4) and, for example, the one by Bird *et al.* [17], this task cannot be fully automatized, especially in an open source context. To avoid bias in our statistical results, we manually resolved aliasing in our data. We started by aggregating on email addresses, to resolve cases with multiple author names. Then, we manually inspected all possible combinations of names and email addresses. One challenge we encountered regards a handful of participants using distinct names and addresses (*e.g.,*'John: `johns@address1.com`', and 'spacej: `spacej@address2.co.uk`'). To resolve these cases, we read the emails sent from these addresses. To answer the research question regarding developers communicating in the mailing list participants (*i.e.,* **RQ3**), we also identified the official committers of the project: We matched names and addresses in our sample with the official list of committers.[6] We also extracted developer user names from the versioning system log. Matching developers was time-consuming, as only few developers use their *[user-name]*`@apache.com` address listed on the Lucene website.

6. `http://lucene.apache.org/whoweare.html`

3.4 What are mailing list participants talking about?

We extracted email data from the Lucene development mailing list,[7] from its inception (Sep 2001) to Nov 2012, totaling 111,366 emails. We aggregated them into threads and removed automatically generated messages. From the resulting 13,019 discussion threads we randomly sampled 1,000 threads and printed the corresponding cards for the card sort. After sorting the first ca. 300 cards, the newly encountered threads started merging in the same groups, reaching a *saturation effect* [64]. To add confidence that the saturation point was reached, and to improve the significance of the subsequent statistical analysis, another 200 cards were sorted, reaching a sample of 506 threads. The remaining cards were discarded.

Through the card sort, 34 groups emerged. During the sorting process, we iteratively gave explanatory names to groups and reflected on how they could be clustered into higher level themes. At the end of this phase, we had clustered the 34 groups into 6 categories and 24 subcategories. We now describe each category and the corresponding subcategories.

(A) Implementation

The implementation category covers the threads related to the implementation of source code artifacts. It comprises topics spanning from proposing new features to be implemented, to discussing implementation details, to contributing with patches. It also includes emails aimed at understanding the system's implementation, or the rationale behind an implementation choice. It comprises four subcategories:

(A.1) **Comprehension:** Participants start comprehension threads to understand (parts of) the implementation, to verify if their knowledge is correct and up-to-date, and to request clarifications on the rationale behind a particular choice (*e.g.,* a used pattern).

(A.2) **Discussion:** Participants initiate discussion threads to ask the opinion of others (*e.g.,* "*what do you think about* [this]"), or to propose one or more possible solutions or ideas (*e.g.,* "*we could do it like* [this]*, or like* [that]"). Usually, discussions revolve around improving an existing code artifact, and start from the comments on a recent feature implementation, bug fix, or submitted patch.

(A.3) **Feature suggestion:** Participants initiate this kind of threads to describe new features from a high-level perspective. Often participants requesting a feature on the mailing list are not directly volunteering to do it, but they mostly propose something for others to do.

(A.4) **Code contribution:** Participants start these threads to let the community know that they have working source code ready to be merged in the system. The code may implement new features; or it may tackle issues that were found by the email author or that were reported in the official bug repository. Contributions are in the form of patches, pull requests, external links, or attached code.

7. `org.apache.lucene.java-dev`

(B) Technical Infrastructure

Most OSS software projects rely on a technical infrastructure to support development, maintenance, and the building process, and to facilitate the communication among project contributors [57]. This category covers email threads related to such an infrastructure; the topics of discussions are (B.1) **building system** (*e.g.,* notification of problems with the building system), (B.2) **documentation** (*e.g.,* decisions on the javadoc), (B.3) **issue tracking** (*e.g.,* move to a new tracking system), (B.4) **mailing lists** (not only the development mailing list itself, but also *e.g.,* the user mailing list), (B.5) **programming language** (*e.g.,* the version of [programming language] to use), (B.6) **testing** (*e.g.,* how to use the continuous testing system), (B.7) **versioning** (*e.g.,* discussions on branches), and (B.8) **website** (*e.g.,* threads on what content to put in the website). Authors of infrastructure threads write to the list for different reasons, such as sending notifications, discussing problems, and posing questions.

(C) Project Status

As described in previous work (*e.g.,* [144, 121, 18]), development mailing lists are also used to raise awareness on the status of the project and to discuss future steps. This category regards this kind of topics, and includes two groups of threads: those about (C.1) **planning** the future development of the project, and those about (C.2) **releases**. Authors of **project status** threads write to the mailing list to announce a new release, decide which issues to fix for a milestone, or discuss the ongoing activity on the project.

(D) Social Interactions

Socializing is an essential ingredient in the long-term survival of OSS projects [51], and mailing lists play an important role in this context [57]. Participants write to the mailing list about the norms, values, and perspectives that are part of the community's operational structure, and to coordinate with others. This category revolves around these social interactions, and threads are about (D.1) **acknowledgement** of efforts (*e.g.,* replying to a code commit to thank the author), (D.2) **coordination** (*e.g.,* raising awareness about an issue in the bug repository, or notifying a participant's absence), greetings and suggestions to (D.3) **new contributors**, and (D.4) **social norms** governing the behavior of mailing list participants (*e.g.,* advices on successfully submitting a patch). Authors of such threads notify their absence, welcome new members, thank someone for a well done bug fix, and tell everyone about newly submitted issues.

(E) Usage

The **usage** category comprises threads with questions and problems about the usage of the software being developed by the programmers enrolled in the development mailing list, and it also includes threads related to external projects. It comprises three subcategories:

(E.1) **Problems and bugs:** Authors ask advice on how to solve issues they have operating the project, or report a general problem they have found. Participants may also bring up a discussion about a problem by forwarding emails sent to other mailing lists, or by answering automatic messages from the issue tracking system.

(E.2) **Information seeking:** Authors write to ask advice on how to complete an operation (*e.g.,* "*How to do* [this]*?*"), on where to find usage related resources (*e.g.,* documentation, examples), and on the right approach to choose among different usage options (*e.g.,* "*What is the proper means to do* [this]*?*").

(E.3) **External projects:** Participants write, for example, to raise awareness about their own, external, software project. They ask to be included among the online list of applications using the main project (*e.g.,* "Powered by"). Participants developing other systems also ask about including their work as part of the main project.

(F) Discarded

This category groups the threads that do not fit into the categories previously described. They are of three kinds:

(F.1) **Auto-generated:** Auto-generated threads, such as emails from the continuous building system or the wiki, that were not filtered out by our heuristics.

(F.2) **Trash:** Threads exclusively composed of unreadable emails (due to formatting problems), and spam emails that are not pertaining to the content of the mailing list (*e.g.,* unsolicited commercial emails).

(F.3) **Turtle:** Email threads that are unrelated to any other thread, or very difficult to classify due to the nature of their content (*e.g.,* meaningless because out of context).

3.5 How often do participants talk about each topic?

Figure 3.3 shows the distribution of the threads among the different categories (see also column 'threads' in Table 3.1).

Implementation is the most frequently occurring category, comprising 36% of the threads. Since the declared aim of the *development* mailing list of Lucene is to be where "*participating developers* […] *meet and discuss issues concerning* Lucene […] *internals, code changes/additions, etc.*", we were surprised that—in reality—implementation threads only count for just a little more than a third of the total threads. This is different from the Linux kernel mailing list (often used for studying developers' interaction), where implementation threads "form the large majority of the traffic on the list." [70]

In comparison, we found the ratio of usage threads in the mailing list to be surprisingly high (27%). In particular, half of these threads regard information seeking (13% overall), in spite of a note on the Lucene website exhorting participants to "not send mail

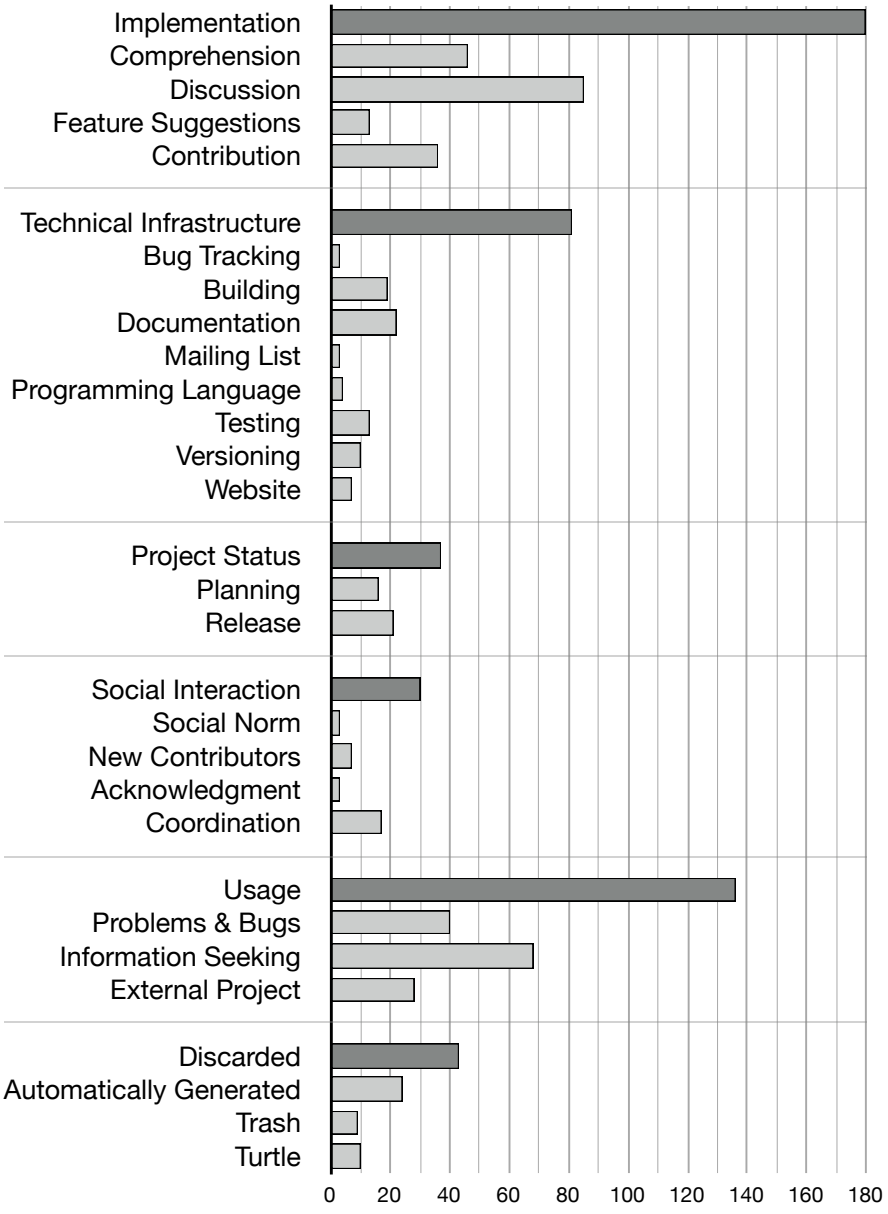


Figure 3.3: *Distribution of threads per category.*

to this list with usage questions or configuration questions and problems.". Moreover, threads regarding PROBLEMS AND BUGS account for 8%. Even considering sampling limitations (see Section 3.9), in LUCENE these threads would corresponds to less than half of the bugs reported in JIRA (up to one fifth, when considering other types of issues), meaning that in LUCENE the mailing list may not be the primary channel for discussing and reporting problems and bugs.

Threads regarding TECHNICAL INFRASTRUCTURE total 16%. The less frequent thread categories are SOCIAL INTERACTION and PROJECT STATUS. It was surprising to us that, despite the mailing list always having been considered the hub for OSS project communication [144, 57], only 7% of the threads in our sample regard the project status, and just 6% regard social interactions among participants.

Finally, there is a not negligible portion (8%) of threads DISCARDED during the card sorting. Besides 10 threads with no clear meaning (TURTLE), and—despite the fact that we performed a rigorous pre-processing and data cleaning phase (Section 3.3)—a substantial amount of noise (7% of the total threads, from AUTOMATICALLY GENERATED and TRASH threads) was still present our sample. We also notice that these threads cannot be clearly distinguished from the other categories: A third of them are replied (*e.g.,* there are threads automatically generated from the wiki, which all have the same subject and thus get threaded), almost a third include developers in the emails (*e.g.,* svn commits initially sent to the mailing list results as sent by the developer author of the commit), and finally almost a fourth of these threads contain code (*e.g.,* svn commits, and change logs from the wiki pages).

3.5.1 How prominent are implementation details?

To better understand how prominent implementation details are, we analyzed the distributions of threads containing code entities (*e.g.,* class names). Are mentioned code artifacts an indication of discussion about implementation details?

In previous work, Bird *et al.* reported that the mailing list is made of more than implementation. They distinguish between *process* and *product*, and use the presence of source code names, such as class names, as classifiers: "Messages that include these source code names are classified as product and the rest are classified as process" [18]. We also apply this distinction to our data and verify whether and how it fits to our categories. We considered the entities mentioned in all the releases of LUCENE, and we analyzed threads to determine whether they contained code entities. Results from our analysis can be seen in the column 'with code entities'.

Our results show that 57% of all the analyzed threads contain code entities, and at least a third of threads in each category contains code entities (except DISCARDED threads, 28%). Of IMPLEMENTATION threads, 77% contains code.

To verify to which degree Bird *et al.*'s classification fits to our data, we first need to define which of our own categories are part of *product*. According to the description, these would

correspond to our IMPLEMENTATION category alone. However, USAGE and DISCARDED threads would not fit in either definition: we decided to include USAGE as *product* (since LUCENE is an API, many usage questions regards its code artifacts), while we consider DISCARDED as *process*.

Our data shows that when only considering threads containing code entities, only 76% of these threads would be regarding *product* (*i.e.,* IMPLEMENTATION and USAGE), while the remaining 24% would actually be about *process*. Moreover, we would only select 70% of all the IMPLEMENTATION+USAGE threads. This is in contrast with findings by Bird *et al.*, where they estimated a correct classification in 90% of the cases.

3.6 Is the development mailing list only for developers?

Once our categories were stable, and after performing several card sort iterations to ensure the integrity of our categories, we resolved aliasing and determined which participants were project developers (*i.e.,* those with commit privileges). Table 3.1 shows the statistical information we collected on the sample of threads categorized in the card sorting process. We include email granularity for completeness.

3.6.1 What do developers focus on?

The overall ratio of threads in which at least one developer participated (column 'with developers') is quite high: Developers are present in more than 75% of the treads in each category, except in USAGE (55%) and DISCARDED (35%). In PROJECT STATUS and TECHNICAL INFRASTRUCTURE threads, developers are present in more than 90% of these threads.

Our results also show that in some categories there is a prevalence of threads 'started by developers'. However, overall, only half of all the analyzed threads have been started by a developer. Developers start the majority of threads in PROJECT STATUS (89% of the threads in this category), TECHNICAL INFRASTRUCTURE, (78%), and SOCIAL INTERACTION (70%). Only 54% of the IMPLEMENTATION threads are started by a developer. This may seem surprising, but, if we look at the subcategories, we can see that only a third of CONTRIBUTION threads were started by developers. This is also due to the OSS structure in general, where a person can be a contributor without committing rights. Participants write to the mailing list *offering* their contributions, hoping that a developer might integrate it in the project. Moreover, users also occasionally write to the development mailing list with program comprehension questions or feature requests.

Furthermore, we notice that only 21% of the USAGE threads were started by a developer, and, in particular, only 4% of the INFORMATION SEEKING threads. It is not very surprising that these threads are not started by LUCENE developers. However, developers also start EXTERNAL PROJECT threads: They often have side projects, built on top of LUCENE, they want to mention in the mailing list (*e.g.,* announcing a new release).

Table 3.1: *Categorization of email threads.*

	categories	threads		replied	with developers	started by developers	with code entities	unique participants		developers	emails		from developers	with code entities
A.1	Comprehension		46	*74%*	*74%*	*43%*	*78%*		66	*39%*		208	*60%*	*72%*
A.2	Discussion		85	*80%*	*86%*	*68%*	*78%*		87	*39%*		551	*70%*	*70%*
A.3	Feature Suggestion		13	*54%*	*77%*	*54%*	*62%*		19	*53%*		35	*66%*	*51%*
A.4	Contribution		36	*75%*	*81%*	*33%*	*81%*		56	*39%*		135	*59%*	*62%*
A	**Implementation (36%)**	**180**		**76%**	**81%**	**54%**	**77%**	**155**		**26%**	**929**		**66%**	**69%**
B.1	Bug Tracking		3	*100%*	*100%*	*67%*	*0%*		8	*88%*		24	*92%*	*0%*
B.2	Building		19	*84%*	*95%*	*53%*	*37%*		25	*56%*		54	*72%*	*24%*
B.3	Documentation		22	*59%*	*95%*	*86%*	*45%*		33	*73%*		78	*83%*	*37%*
B.4	Mailing List		3	*33%*	*67%*	*67%*	*0%*		4	*75%*		4	*75%*	*0%*
B.5	Programming Language		4	*100%*	*100%*	*75%*	*50%*		27	*52%*		100	*54%*	*18%*
B.6	Testing		13	*77%*	*92%*	*92%*	*62%*		21	*81%*		71	*94%*	*42%*
B.7	Versioning		10	*80%*	*90%*	*80%*	*20%*		28	*61%*		76	*78%*	*4%*
B.8	Website		7	*86%*	*100%*	*100%*	*0%*		13	*85%*		32	*94%*	*0%*
B	**Technical Infrastructure (16%)**	**81**		**75%**	**94%**	**78%**	**36%**	**76**		**43%**	**439**		**77%**	**21%**
C.1	Planning		16	*88%*	*94%*	*88%*	*56%*		48	*54%*		233	*84%*	*19%*
C.2	Release		21	*71%*	*90%*	*90%*	*38%*		34	*56%*		126	*85%*	*28%*
C	**Project Status (7%)**	**37**		**78%**	**92%**	**89%**	**46%**	**63**		**48%**	**359**		**84%**	**22%**
D.1	Social Norm		3	*33%*	*100%*	*100%*	*0%*		4	*75%*		6	*83%*	*0%*
D.2	Contributors		7	*71%*	*86%*	*71%*	*29%*		15	*80%*		26	*85%*	*19%*
D.3	Acknowledgment		3	*0%*	*100%*	*100%*	*33%*		3	*100%*		3	*100%*	*33%*
D.4	Coordination		17	*35%*	*65%*	*59%*	*47%*		17	*47%*		29	*41%*	*38%*
D	**Social Interaction (6%)**	**30**		**40%**	**77%**	**70%**	**37%**	**30**		**57%**	**64**		**66%**	**27%**
E.1	Problems & Bugs		40	*58%*	*70%*	*35%*	*80%*		53	*34%*		128	*45%*	*81%*
E.2	Information Seeking		68	*68%*	*47%*	*4%*	*60%*		99	*24%*		210	*37%*	*61%*
E.3	External Project		27	*59%*	*52%*	*41%*	*30%*		45	*36%*		86	*52%*	*24%*
E	**Usage (27%)**	**135**		**63%**	**55%**	**21%**	**60%**	**164**		**20%**	**424**		**43%**	**60%**
F.1	Automatically Generated		24	*33%*	*25%*	*21%*	*29%*		6	*50%*		156	*5%*	*19%*
F.2	Trash		9	*33%*	*44%*	*44%*	*11%*		16	*63%*		30	*77%*	*20%*
F.3	Turtle		10	*60%*	*50%*	*30%*	*40%*		19	*42%*		27	*44%*	*33%*
F	**Discarded (8%)**	**43**		**40%**	**35%**	**28%**	**28%**	**34**		**44%**	**213**		**20%**	**21%**
	Total	**506**		**67%**	**73%**	**50%**	**57%**	**315**		**16%**	**2428**		**63%**	**46%**

3.6.2 Dynamics of Interactions

By analyzing the population of mailing list participants, we found that only 16% of the participants are official committers (column 'developers'). Thus, the vast majority of participants in the development mailing list are *not* LUCENE developers. We asked ourselves: How are participants interacting via the mailing list? Do developers have a particular position?

The column 'Unique participants' indicates the number of individual people participating to discussions threads. When the number of participants is lower than the number of threads, this means that people are participating in more than one thread (*e.g.,* this is the case in the IMPLEMENTATION category). Similarly, a higher number of participants than the number of discussion threads indicates "one-timers" (we observe this in the USAGE and PROJECT STATUS categories).

To better understand where participants interact, we counted threads that are *replied to* (*i.e.,* with more than one email). The analysis of the replied threads by category gives an idea of the responsiveness of the mailing list and the "rhythm" of talks within each category. Interestingly, the least responded threads are those about the SOCIAL INTERACTION (40% overall). We also analyzed multi-email threads in terms of first-response rate (*i.e.,* how long before the first reply). Threads are answered within a day: TECHNICAL INFRASTRUCTURE and SOCIAL INTERACTION threads get faster reactions (first reply within two hours), while IMPLEMENTATION and USAGE threads might take up to 21 hours to be replied to. We did not find a statistically significant difference in responsiveness depending on who sent the first email (*i.e.,* a developer or not).

3.6.3 The Overall Picture

Figure 3.4 puts all the threads in our sample in a nutshell: It shows, by category, how many threads are with *vs.* without code entities (right *vs.* left side), with *vs.* without participating developers (bottom *vs.* top bar), and how many of the latter were initiated by a developer (dark *vs.* light color). The exact amount of threads is reported for each "type".

We see a large amount of threads without developers in the USAGE category compared to other categories, a prevalence of developers on IMPLEMENTATION and TECHNICAL INFRASTRUCTURE, and a large amount of threads without developers in the USAGE category compared to other categories.

3.7 What is the role of the development mailing list?

By answering the previous research questions, we found that the official description of the aim of the development mailing list does not correspond to its real usage. Our fourth research question seeks to understand the role of development mailing lists for the communication in OSS at large. We attempt to achieve this by triangulating the information

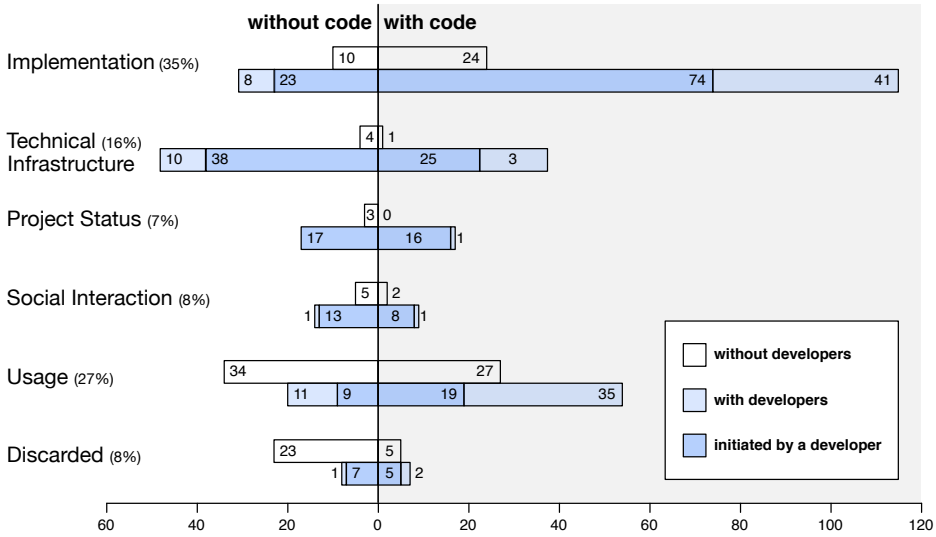


Figure 3.4: *Thread types distribution: gives an idea of the distribution of threads and the different "shapes" of our categories.*

that we obtained by reading the 506 threads during the card sort, by analyzing the statistical data on the categories, and by searching more facts in the rest of the mailing list.

3.7.1 Is in the mailing list where all the communication occurs?

Previous literature stated that mailing lists are "the bread and butter of project communications" [57], and in particular that "the developer mailing list is the primary communication channel for an OSS project" [70]. Reading the analyzed emails, however, makes it clear that the development mailing list is just *one* of the communication channels used in a OSS project; in fact, other channels also play an important role:

Issue Repository – Many threads provided evidence that a significant amount of communication takes place in the Jira issue repository: Participants often reference Jira issues in emails, or omit details because already mentioned in the issue discussions. Altough project members started using Jira only in mid 2005, in our entire population of emails (Sep 2001 to Nov 2012), we found 69,632 (63%) messages automatically forwarded from discussions taking place in Jira, still showing a clear increasing trend in its usage.

IRC – Participants talk about the project and implementation details also on the development IRC channel (created in Apr 2010): "*I propose that we chat on irc at*

#lucene-dev [...]. *I'd like to discuss the core elements of the Spatial Strategy API, namely makeQuery,* [...]*, and SpatialOperation.*"

User Mailing List – The user mailing list also plays a role in the project and developers' communication. Developers monitor it, for example, to understand the usage of the system (*e.g.,* "*I am wondering if TermVectorsWriter is still used* [...]. *The reason I am asking is the java-user* [email subject]"), to improve the documentation (*e.g.,* "*about the exposure of FieldCache in the documentation* [...] *see for instance this discussion in the user list*"), and to forward interesting discussions to other developers.

In person – We found evidence that developers also have a number of in-person meetings where they discuss about project details (*e.g.,* "[Developer] *and I talked a little bit about this at the ApacheCon*").

3.7.2 Is the mailing list for driving coordination?

Previous work reported that a portion of the communication taking place in the mailing list regards coordination between developers as they work together on the software [144, 18]. Surprisingly we found a very small amount (3%) of COORDINATION threads, with an average of less than two emails per thread; moreover, most of these threads were not for fostering collaboration on the implementation, but for raising awareness on already accomplished work.

By reading emails, we found evidence that developers, instead of using the mailing list, prefer to coordinate through items in the issue tracking system. For example, one developer who sent an email with: "*If you can help, please coordinate here on this thread, so that we don't stomp on each other.*" afterwards corrected himself in a second message: "*Sorry, should have said, please coordinate on the* JIRA *issue*". Another developer, who was guiding a newcomer through the coordination norms in the project, wrote: "*You will not fall out of sync in short order, especially if you work with* JIRA *so others know what you are doing.*"

In addition to the issue tracking system, developers also coordinate in the IRC channel: "*As we discussed on IRC yesterday, the number of people* [...] *qualified to write* [code] *will still be very small*"; or in person: "*I talked about this with* [list of developers] *in Berlin, and they all like this proposal.*" Moreover, developers remain coordinated by keeping track of code changes. They do this by reading emails generated by the versioning system, sometimes forwarding these emails to the development list along with their comments.

3.7.3 Is the mailing list used for peer code review?

Rigby *et al.* reported that OSS mailing lists are used for submitting patches and performing peer code reviews [145]. We indeed found that most patches led to a purely technical discussion, while some also led to a discussion of project objectives, scope, or politics.

The vast majority of threads with patches in our sample was sent earlier than the introduction of the JIRA issue tracking systems: After mid 2005, we saw the number of patches drastically diminishing. Reading emails, we found additional evidence that patches, nowadays, are not sent anymore to the mailing list, but they are sent, discussed, peer-reviewed, and approved/rejected in the issue tracking system. For example, when a contributor asked to go to the issue repository to review a patch: "[issue id] *Did anyone try out or took a look at my redesign* [...]*? I'd love some feedback.*" A senior developer explained: "*You should submit *all* patches you want to commit to* JIRA *first to give others the chance to review and possibly vote against the patch.*" This finding is inline with the project website: "*How to contribute:* [...] *Finally, patches should be attached to a bug report in* JIRA."

3.7.4 Is the mailing list the hub of project communication?

Although other researchers also found that the development mailing list is not the only channel of communication in OSS projects (*e.g.,* [18, 150]), it has always been considered the *hub* of project communication. For example, Mockus *et al.* reported that developers use "email lists exclusively to communicate with each other" and that "due to some annoying characteristics of the [issue tracking system], very few developers keep an active eye on [it]" [121].

When more communication repositories exist, the policy of most OSS projects is to transfer all the official decisions and useful discussions to the mailing list [57], so that they can be later retrieved. These traceability links between the development mailing list and other communication repositories must be manually created and updated. We found some cases in which the traceability link was established, but, more often and in line with the findings of Sarma *et al.* [150], we found a clear *disconnection* among repositories, which led to coordination issues and duplicated/lost information. For example, because of multiple communication repositories, developers need to raise inter-repository awareness (*e.g.,* "*I submitted a patch for* [JIRA issue] *a month ago,* [...] *it hasn't been picked by anybody yet*"), ask where a discussion takes place (*e.g.,* "*were there emails about it or it has been discussed on IRC?*"), and go back and forth between the same discussion taking place in more venues (*e.g.,* "*We would like to implement* [this]*, which was discussed in* JIRA"). Overall, our investigation provides evidence that the various communication channels work in parallel, remain disconnected between one another, and the development mailing list does not play (anymore) the role of a hub.

3.8 Implications

From our investigation, we found that the role of the development mailing list, previously considered as the place for discussing code artifact implementation and as the hub of all project communication, has changed. In the following we describe some of the implications deserving future research.

On Communication. Communication is scattered among repositories. This once again underlines the importance of adopting a holistic view and considering software repositories as a whole, not only in research but also in practical development. In fact, even project developers have problems in maintaining awareness of each other's work in the current situation.

Automatically recovering traceability links among communication repositories would free developers from the task of recovering scattered traces of previous communication, and would help researchers having a more complete picture of the development process. More tools for maintaining awareness would be also necessary to improve developers' productivity. Since the advent of better issue tracking systems led to a shift in the habits of OSS participants toward different communication means, we should investigate the features in issue tracking systems that produced this change of direction.

On Data Quality. Different communication topics take place in the development mailing list; to extract valuable information we have to take this into account. We have to improve our methods for removing noise (8% of our sample, even after a careful pre-processing phase), then there are the premises for future work on automatic classification of threads of discussions, so that only the relevant categories would be taken into account for analysis. We also underlined the importance of a correct aliasing resolution, which still cannot be fully automatized. We provide our complete aliasing and thread categorization to benchmark novel automatic techniques. Nevertheless part of the communication data is going to be lost, because we found that communication takes place in unrecorded places, even in OSS systems. We have to take this into account in our statistical analyses.

On Software Development. Not only committers respond to the development mailing list, but also other people are very active. We could consider techniques for finding code experts not only among active contributors, but also among active respondents of the mailing list. Moreover, considering the shift to other communication repositories, mailing list may not be the right venue for studying code review anymore.

3.9 Limitations

One potential criticism is that a case study with one project may provide little value. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science [56]. To understand mailing list communication we read emails spanning 11 years of mailing list usage and written by 155 diverse participants. To answer our research questions, we also analyzed data from the code repository, the project website, and email threads external to our sample.

To ensure that the thread categories emerged from the card sort were clear and accurate, and to judge whether our set of category provides an exhaustive and effective way to organize mailing list communication, we conducted a validation phase that involved three people external to the pair-card sort. Three software engineering researchers conducted a

closed card sort on 50 cards (10%) randomly selected from our sample. They observed that the 6 main categories were clear and covered all thread topics. We measured inter-rater agreement: The Fleiss' Kappa value for the four ratings of the random sample was 0.657 (*i.e.,* substantial agreement) for the six categories, and 0.505 (*i.e.,* moderate agreement) for the 24 sub-categories (which were more difficult to be all recalled by participants). To verify whether there was a systematic error in our catalogue, we also measured the inter-rater agreement among the three experiment participants. Their agreement was 0.592 for the main categories, and 0.458 for sub-categories (both corresponding to a moderate agreement, suggesting there was no systematic misinterpretation).

Threats to validity: Concerning *internal* threats, the sample size (506) of threads provides a 98% confidence level and 5% error on subsequent estimations of proportions [170]. Concerning *external* threats, other OSS projects use communication tools similarly to Lucene, for example, 87 other Apache projects are also using the Jira issue tracking system and have IRC channels. However, team dynamics may differ and our research should be repeated in other contexts.

3.10 Concluding Remarks

We investigated the communication taking place in OSS development mailing lists, finding that email threads cover a range of topics and that communication on implementation is only a portion of them. We found that code artifacts are also mentioned in topics not related to implementation, and that project developers are not the majority of the participants. We established that the development mailing list is only *one* of the communication channels used in an OSS project, and we found evidence of a shift in the communication habits with an increased usage of the issue repositories.

The entire dataset used in the experiment, including the cards, the resolved aliases, and detailed statistical results (by thread and by category), can be found online on the supporting website: `http://www.st.ewi.tudelft.nl/~guzzi/oss-communication`.

Part II

Exposing Information

4 CARES: Relevant Engineers

Enterprise software developers must regularly communicate with one another to obtain information and coordinate changes to legacy code, but find it cumbersome and complicated to determine the most relevant and expedient person to contact. This becomes especially difficult when the relevant person has transferred teams or changed their personal contact information since contributing to the project.

In this chapter, we present a year-long series of surveys and interviews we conduct to help us learn how, why, and how often software developers discover and communicate with one another. Following what we learn, we design, deploy, and evaluate a domain-specific, IDE-embedded, photo-oriented, communication tool called CARES*: Colleagues and Relevant Engineer's Support. After deploying our tool, iteratively refining it, and deploying it again on a company-wide scale, most users report that it simplifies the process of finding and reaching out to other developers, and that it offers them a sense of community with their colleagues, even with those colleagues not currently working on their team.*[1]

1. This chapter is a blended version of the paper "Facilitating Enterprise Software Developer Communication with CARES" [76], published in the proceedings of the 28th International Conference on Software Maintenance (ICSM 2012), with authors Guzzi, Begel, Miller, and Nareddy; and the paper "Facilitating Communication between Engineers with CARES" [75], published in the companion (tool demonstration track) proceedings of the 34th International Conference on Software Engineering (ICSE 2012), with authors Guzzi and Begel.

4.1 Overview

Successfully developing and maintaining software products requires effective communication between dependent engineers. Our research explores the challenges of communication at Microsoft, where software products are extremely large, long-lived, and developed by non-collocated product teams. Software engineers must often maintain software written by developers who have left Microsoft or moved to other teams. They try to rely on specifications, documentation, and source code to answer their questions, but in the end, engineers prefer to speak with knowledgeable experts or people with the authority to coordinate actions they need to get their work done [59, 103, 162, 125]. Unfortunately, when there is too little information shared between dependent engineers about the project's status and changes, the work relationships required for fluid collaboration suffer and threaten the project's success [13].

Grubb and Begel studied the causes of the paucity of inter-team communication and found that software developers felt inhibited from sharing information on their work [68]. With others on the same team, developers would communicate about their work quite openly, but with people in other teams, products, and divisions, they would share less. When asked, engineers reported an aversion to "spam" other engineers with work notifications they might not be interested in. The asymmetry of dependencies on software teams and the modular boundaries induced by their software architecture [42] sometimes prevent engineers from noticing that others depend on their work. If an engineer thought no one cared about their status or changes, he would not think to communicate at all.

We tackle this problem from the reverse perspective, looking at the software developer who needs to communicate with a code owner who can explain some aspect of the software, the rationale behind it, or who had the authority to coordinate joint action to improve it. Over the course of a year, we conducted surveys and interviews to better understand how and why Microsoft software developers communicate with one another, and how often they do so. We discovered the criteria developers use to identify and choose a set of *relevant* people, how they select the most *expedient* person to contact, the means by which they contact that person, and how often their conversations led to positive working relationships.

We then designed, developed, deployed and evaluated a new communications tool to encourage developers to communicate with one another and simplify the process of doing so. Our tool, CARES: Colleagues and Relevant Engineers' Support, is a Visual Studio extension designed specifically for software engineers who want to communicate with others about source code. CARES displays a context-sensitive array of photos of the engineers who are most tightly connected to the code in each file currently being edited in the IDE. To help developers select the best person with whom to communicate, each photo has a tooltip that reveals the person's code history, organization, physical location, and current availability. Developers can then choose to meet the person face-to-face (F2F), or initiate contact using email, instant messaging (IM), A/V chat, application sharing, or screen sharing buttons right in the tooltip.

Structure of the chapter. We first describe our year-long study (Section 4.2) and results (Section 4.3), then we present CARES (Section 4.4) and its evaluation (Section 4.5). Finally, we show how our studies and tool were inspired and influenced by the research literature (Section 4.6), and conclude with our suggestions for researchers interested in developing usable software enterprise-oriented communication tools (Section 4.7).

4.2 Methodology

In this section we present the methodological steps we followed to (1) better understand how and why Microsoft software developers communicate with one another, and how often they do so; (2) discover the criteria developers use to identify and choose a set of *relevant* people, how they select the most *expedient* person to contact, and the means by which they contact that person; and (3) (iteratively) derive requirements for the design and refine the implementation of a tool to facilitate developers' communication.

Survey 1 – Over a two week period at Microsoft in July 2011, we conducted a 50 question web-based survey (Survey 1) divided into 4 sections: demographics and three communication scenarios (described below). The questions were drawn from Begel *et al.*'s previous study of inter-team coordination [13] and were piloted with several developers before being deployed. Out of 500 randomly sampled Microsoft developers (5% of all developers) invited to take Survey 1, we received 94 valid responses (19% response rate). Invitees were incented to respond by a raffle for US$100. Demographically, the 94 respondents of Survey 1 had spent an average of 11.3 years (SD=7.5) in the software industry and 6.9 years (SD=5.2) at Microsoft. Most (68%) reported that they had previously worked at other companies. 97% of respondents were developers, and 90% of those were individual contributors (ICs).

Survey 2 – After the responses to Survey 1 were received, we realized the need for some additional questions on communication frequency. We sent a supplemental survey (Survey 2) to respondents of the first survey who indicated they were willing to speak further on our topic. Survey 2 was sent to 32 respondents of the first survey, and we received 18 valid responses (56% response rate).

CARES Design and Implementation – From the research literature, our own prior research, and what we learned from Survey 1 and 2, we decided to build a tool to help with the problems that developers reported. We designed the CARES tool to help developers find and select the most relevant person to communicate with for their needs. To make that selection actionable, CARES supports initiating communication with the selected party.

Deployment 1 and Interviews – We deployed CARES to the 32 people who received Survey 2 and gave them instructions on how to install and use it. After a few weeks, we emailed these pilot users and asked them if they would be willing to be interviewed about it. Eight users agreed after having used CARES from between one and three weeks. They were all developers (including two developer managers) and worked in

six different Microsoft departments. Each worked on a team with 4 to 9 people and regularly collaborated with 3 to 30 engineers working on other teams. The first two authors of this research interviewed each of those eight CARES users for an hour; one author asked questions and engaged the interviewee while the other recorded the interview and took copious notes. The interviews were transcribed verbatim when it improved the accuracy of the notes. In each interview, interviewees were asked to demonstrate their use of CARES in their own workspace.

Deployment 2 – From the lessons learned from the first deployment, CARES was further developed, while continuing to make the tool available to 30 pilot users (out of the 32 who received Survey 2). Feature and user interface enhancements were added to improve CARES' compatibility and robustness with the Microsoft's development environments. In March 2012, CARES was publicized at an internal Microsoft research conference and made available to all employees at the company. To monitor CARES usage, the tool included a logging facility to report feature usage.[2] Until end of June 2012, CARES has been used by a total of 106 employees (excluding the original 30 pilot users and anyone associated with the development of the CARES tool). CARES continued to be used (defined as at least twice in the last two weeks of June 2012) by 36 of those 106 users. We recorded a total of 4,943 log sessions. The most prolific user used CARES 411 times since installation.

Survey 3 – In April 2012, we conducted a third survey (Survey 3) of 87 (at the time) known users of CARES. In addition to demographic questions, we asked about their usage of the CARES tool, their understanding of the visualization, their perception of its utility and impact on their daily work, their subjective assessment of its features, and their suggestions for improvement. We offered no incentive to answer this survey. We received 24 responses (28% response rate), of which 19 were developers (79%) and 5 were testers (21%). Respondents spent 9.1 years (SD=6.9) in the software industry and 4.8 years (SD=4.1) at Microsoft. The log data shows that 10 of these survey respondents still regularly used CARES when this research was completed at the end of June 2012.

4.3 Developer Communication

One of our first research questions was to identify, in detail, why and how developers communicate with one another about source code. Previous studies indicated that asymmetry of dependency was an important factor in mediating the quantity of communication [68], so we asked study participants questions that concerned both a forwards and backwards version of communication scenarios we included in our surveys.

2. File paths opened in the editor and employee names and email addresses visible in the UI of the tool are hashed before being written to the log. This identifying information is not necessary for our analyses of the data. Logs are collected automatically, but users can opt out in a Settings dialog.

The first survey scenario (Sc1) asked respondents to consider the *most recent time* they needed to communicate with someone on another team about source code they saw in their IDE.[3] The second scenario (Sc2) asked about the most recent time someone on another team asked them about code they wrote. The third scenario (Sc3) asked about the most recent time someone on another team had asked them about code they did not own at the time. Table 4.1 lists the reasons (drawn from our previous studies and the research literature) respondents could check off to explain why they needed to communicate. In this report, we divide them into three categories: *coordination* (*i.e.*, communication requiring negotiation or extended interaction), *seeking information*, and *courtesy* (*e.g.*, notifying someone that you are about to change their code).

Table 4.1: *Why respondents communicate about source code, divided by category, with response rate for each survey scenario: Sc1 (N=91), Sc2 (N=84), Sc3 (N=69). Bolded reasons are in the top four over all. Bolded numbers indicate for which scenario the reason was chosen most.*

Reasons for communication	Resp. [%]	Sc1	Sc2	Sc3
Coordination:				
Discuss a change I want to make to the code		**54**	36	26
Know if my use case was supported by the code		37	**42**	29
File a bug on the code		**33**	11	12
Know if a bug on the code was fixed		**22**	14	9
Propose a collaboration on a topic related to the code		12	**23**	10
Take ownership of the code		**9**	8	N/A
Ask [the respondent] to make a change to the code		N/A	**20**	14
Seeking Information:				
Ask how the code worked		51	**65**	**65**
Ask why the code was written that way		**47**	33	29
Find out who wrote the code		**15**	4	12
Ask if the code had test cases		**8**	2	4
Learn more about the code because I used to work on it		**7**	1	4
Courtesy:				
Ask permission to make a change to the code		**20**	13	10
Let them know I filed a bug on the code		**16**	11	7

The most frequent reasons, marked bold, are discussing a code change, inquiring about support of a particular use case, and to learn how the code worked and why it was written that way. The top three reasons were highly correlated with one another — about half of respondents reported two of the three, while 13 reported all three. These reasons are similar to those found in previous studies [110, 103]. The diversity of responses is interesting as well, as more than 80% (Sc1: 80% (N=90), Sc2: 91% (N=82), Sc3: 81% (N=67)) of respondents indicated that the specific conversation they referred to in their responses was typical for them.

3. We ask about a specific event to avoid memory and generalization biases by respondents.

After deploying CARES, we expected that users would be inspired to ask us to extend its functionality. Survey 3 respondents asked for a way to easily see and communicate with other sets of people than just those who had made checkins. The most popular sets were developers who had code dependencies (*i.e.,* called methods, used classes, etc) (36% N=22), and testers who wrote tests for code in the file (36% N=22). Some also wanted to see anyone who had ever made changes to any file in the same Visual Studio project or solution (18% N=22), and anyone who ever reviewed a checkin to the file (14% N=22). This indicates the need to investigate additional communication reasons and usage scenarios in the future.

4.3.1 Finding, Selecting, and Contacting a Relevant Person

We anticipated that when a developer wants to talk about code, he would have a difficult time finding someone relevant to talk to. First, he must discover the set of possible choices of appropriate people, and then pick the one he believes is most relevant and expedient for his needs. To our surprise, the majority of respondents (66%) said "it was easy to find someone relevant to communicate with." Only 11% said it was not. However, in our interviews, the developers complained that finding the right person was a tedious process. We found that this process involves many factors.

Table 4.2: *Reasons for choosing a particular a person to talk to, with response rate for each survey scenario: Sc1 (N=91), Sc2 (N=83), Sc3 (N=69).*

Why pick this person?	Resp. [%]	Sc1	Sc2	Sc3
I thought they owned the code		56	**63**	48
Their team owned the code		52	**54**	45
They contributed to the code		45	**60**	29
I already knew them		26	N/A	**28**
I knew one of their teammates wrote the code		13	N/A	**30**

In Survey 1, we asked respondents how they choose a relevant person. In all three scenarios (see Table 4.2), they indicated that code ownership and contribution are the most important criteria. 26% (Sc1: N=91) said, if possible, they would pick a person they already knew. In our interviews, we explored this process further. All eight interviewees said that they search the source code checkin history first. From each checkin, they find the author's (committer's) email address, the explanation for the checkin, and the diff showing what changed. One interviewee explained that the owner is the person to contact because "*ownership is important for understanding design rationale.*" On some teams, the owner is the committer with the most checkins. On others, "*there is no single owner because a lot of people touch every file.*" A third group of teams contacts the author of the most recent change, instead of the owner: "*I look at the* [source code] *history and find out who last modified the file...*[The] *last person makes more sense than number of times.*" Each developer's rules appeared to be team-specific, even among those working on different

teams for the same product. This fits in with Microsoft's grass-roots software process culture, in which each team is encouraged to use whatever process works best for them, so it "*would be hard to get the real owner based on what* [my] *team*['s] *practice is.*" Of course, if they see someone they already know, developers may pick that person over everyone else. "*The closer to me the dev is, the easier it is* [to talk to him]." We found it surprising that no interviewee ever spoke about searching for a subject matter expert about the code. Perhaps they expected that the information they sought could always be provided by anyone with knowledge about the file.

After finding a person, the next step is to contact her. Many survey respondents indicated that they initiated contact using multiple communication channels at the same time. Over all three scenarios, email was most common, followed by face-to-face contact, and IM (Sc1 (N=91): Email 86%, F2F 23%, IM 20%. Sc2 (N=83): Email 76%, F2F 31%, IM 22%. Sc3 (N=69): Email 72%, F2F 30%, IM 20%). Other channels were rarely used.

The choice of communication channel often depends on how well the developer knows a person, whether the person is currently available online, and how far away he is. One interviewee explained, "*I mostly use email and IM. But, if they are on the same floor, sometimes I'll visit them in person. If I know him, I'll IM him.* [But] *if he's far away, I will ping or email. If I don't know him, I will email him.*" Another said, "*I would IM or email. If* [they are] *not online, then I would email. Even if* [they are] *in the building, I will IM anyway. When they get back (if* [they were] *away) I will IM them back. If* [they are] *out of office, then I will send email.*" The person's job level also has an impact: "*If* [they are] *low on the management chain, I talk to them in person. If high up in the chain, I send an email. It doesn't matter if I know them.*" Another interviewee who felt similarly explained. "*Usually managers are busy... I'd send an email before* [I would] *knock on their door.*" When contacting a manager, they said they would ask simply for a more appropriate IC contact.

Thus, although developers and their teams have distinct "algorithms" for finding and selecting the most relevant and expedient person to speak with, overall, they seem to consider at least six criteria: ownership, checkins (most recent or most numerous), a preexisting relationship, physical distance, job level, and online availability.

The interviewees communicated most often with their own teams, all of whom were collocated on the same floor of their building, and frequently in the same hallway. However, conversations with developers on other teams occurred often as well; in Survey 2, respondents indicated that they contact other developers two to five times per work task, and that most contact someone at least once a day when doing coding tasks. Most responses to emails or IMs occurred immediately (31% N=87) or while the asker was still working on his task (56% N=87). Though we thought it would be more difficult to get a response from a file owner on another team, 73% (N=91) of Sc1 respondents said that they were happy with the timeliness of the response they received.

These kinds of conversations resulted in mainly positive impressions. Survey 1 respondents reported being satisfied with conversations they had for all of the scenarios (Sc1: 84% (N=91), Sc2: 94% (N=84), Sc3: 90% (N=69)). In Scenarios Sc1 and Sc2, most re-

spondents agreed with the statement, "*the outcome of the conversation we had is still relevant for me*" (Sc1: 81% (N=90), Sc2: 83% (N=82)). Such frequent, positive, material communication is likely to help engineers build and maintain positive working relationships with one another.

4.4 Tool Design and Implementation

In this section we describe CARES, our tool to support developer in finding, selecting, and contacting relevant colleagues, and the factors that influenced its design, many of which originated with Grudin's studies of groupware failures [69].

CARES is an IDE extension that displays a vertical array of *photos of engineers* in the whitespace in the upper-right corner of the current editor window.[4] To help developers select the best person with whom to communicate, each photo has a tooltip that reveals additional *relevant information*, such as the person's code history. Developers can then choose the best way for them to *contact a person*. There is a US patent for CARES [12].

Photos of engineers: The photos shown are context-sensitive; in each editor, CARES shows the faces of those engineers who contributed to the file (*i.e.,* checked in changes on any branch). This minimizes the set of people developers have to consider speaking with about a topic related to the file's source code. This reduction can be significant, as product teams often employ hundreds of developers, and anyone, on any team, may have worked on the code in the past. Some of the interviewees worried that a few files in their product were edited by many colleagues, and they would not be able to see them all. During Deployment 1, we asked the interviewees to show us one of their own files that had many committers, but none could find any with more than five people. When some Deployment 2 users also complained about too many photos, we added the ability to switch to a more vertically compact, name-only view. Incidentally, this name-only view addresses a potential problem among those developers prone to making stereotypical judgments based on seeing someone's race, ethnicity, age, or gender in a photo. CARES respects employee's choice over how their photos are used by observing the Microsoft IT-standard opt-out option.

Relevant information: When the developer hovers over a person, a tooltip displays her name, email address, title, department, manager (because managers are often more widely-known than ICs), office location relative to the user (because developers are more likely to walk down the hall to meet with someone than walk up and down the stairs), and a colored bar indicating her availability (taken from her IM and work calendar status). The tooltip also shows her historical contribution to the code: whether she made the most recent checkin, made the most checkins of all of the people shown, or added the file to the repository, and how many commits relative to the others she made to the file. Finally, the tooltip reveals the dates of her first

4. Source code typically has a lot of whitespace on the right margin.

and most recent checkins, enabling the user to figure out if she is currently working on the code, or has moved on.

Contacting a person: Once the developer chooses a person, (s)he can click on email, IM, A/V chat, Visual Studio application sharing, or screen sharing buttons in the tooltip to initiate contact. CARES helps contextualize the ensuing conversation by filling in the current file path, class, and method (if applicable) as the message subject.

In the following, we present a walk-through of CARES, showing how the tool can help developers discover and contact relevant colleagues to help find information and coordinate action. Subsequently, we present how CARES was implemented from a technical point of view, and we report on some deployment considerations regarding the tool's extensibility and adoptability. Finally, we show how we overcame the challenge of linking the email address of a committer of a checkin to data describing who that employee is for a long-lived codebase.

4.4.1 CARES Walk-Through

For this walk-through, we introduce Jane, a fictitious character inspired by the developers we interviewed, and follow her as she performs one of her tasks, supported by the CARES tool. Jane is an experienced developer who has just changed teams for the first time in three years after successfully shipping her last product. She is assigned the job of designing a new architecture for a common infrastructure platform to support her new team's long-lived suite of internal software tools. To design this properly, she must find and understand the requirements, scenarios, implementations, and bugs of all the old tools. Various members of her new team have been helping her get started by sending her pointers to the top-level directories where each tool is located.

1. Jane opens an email with a URL in it that says very simply, "This directory contains the code for the IT administration tool." Jane is not sure what this tool really does, but she points her IDE at the directory in the repository, checks out the code, and opens up the project. Jane thinks to herself, "where should I begin?" Perhaps she can discover something about this tool's methods from the filenames. She sees one file that is suggestively named, `RecommenderAdapter.cs`, and decides to open it.

   She sees a `GetRecommendations()` method that uses library methods she has never seen before (Figure 4.1). "What could this be for?" Jane wonders. "Maybe I can find a person who knows enough about the design to help me understand its purpose."

2. In the upper-right corner of `Recommender-Adapter.cs`' editor window, she sees the CARES visualization, which shows the photos of three people, a woman and two men (Figure 4.2). She knows the woman, but has never met the two men. She hopes that one of the three might be able to help.

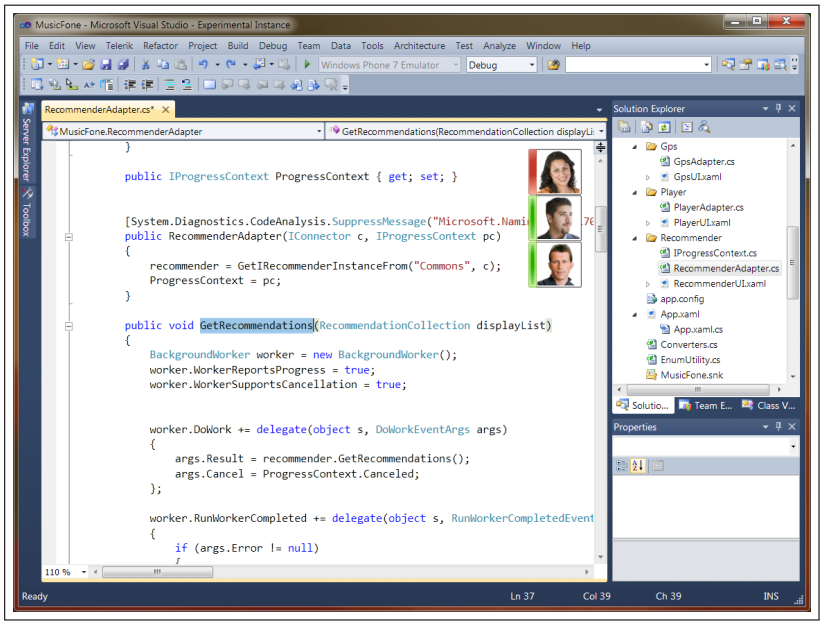


Figure 4.1: *Janes sees a GetRecommendations() method.*

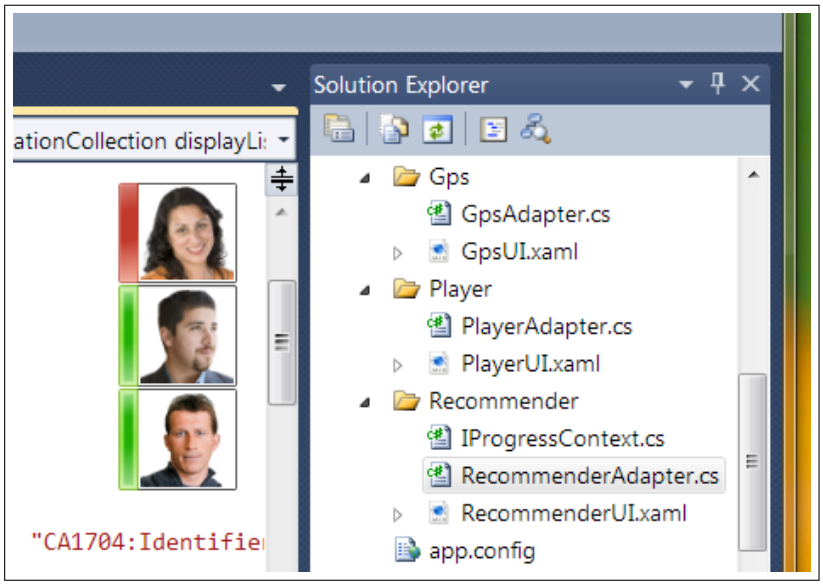


Figure 4.2: *Janes looks at the CARES visualization.*

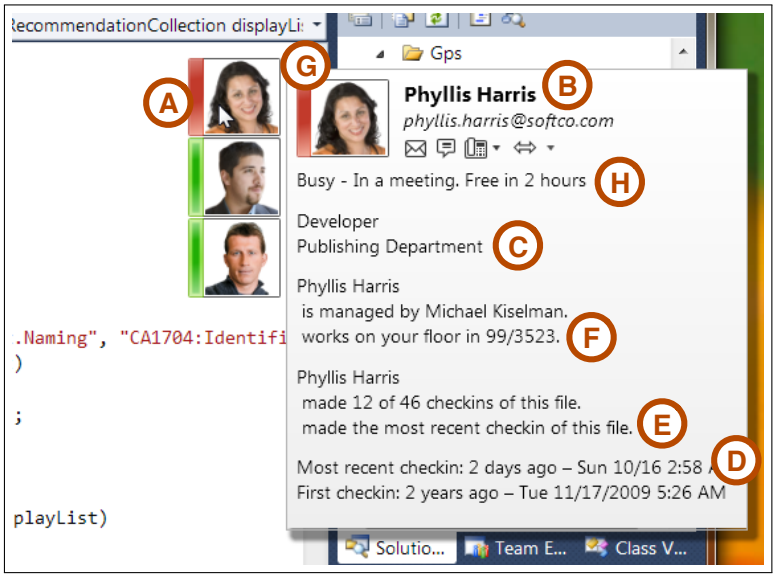


Figure 4.3: *Jane takes a closer look at Phyllis.*

3. She figures that her friend, Phyllis, has worked on the file most recently (CARES sorts people's photos, with the person that made the most recent checkin on top), and hovers her mouse over her photo (A in Figure 4.3). A tooltip shows that she is Phyllis Harris (B), a developer in the same Publishing Department (C) that Jane works in. Phyllis worked on the code only two days ago (D), and is the most recent person to make a checkin (E). "Surely, Phyllis must know what this function is doing." Even better, Phyllis works nearby on Jane's floor (F) and it would be easy to walk over to meet her in person. But, Phyllis' presence indicator is red (G); she is in a meeting (H) and unavailable to help right now.

4. Jane looks at the next photo, David Pelton, another developer in the same department. David works for Jill (I in Figure 4.4), one of Jane's friends since she started working at the company. David worked on the code last week, but he has not done much (J); "he probably does not yet have a complete understanding of the code."

5. The last person listed is Jon Cantrell, a developer in the IT department. He works in Houston (K in Figure 4.5), far away from Jane, and Jane does not know anyone in the IT department. Jon added the file to the repository (L), and made the most checkins of that file (M). Jane believes that he was the original author of the codebase, however he has not touched this code in three years (N). Jane concludes, "Since he has moved on to a different part of the company, he probably transferred ownership of the code to someone else. I am sure he would rather not be bothered about this unless it was absolutely necessary."

Figure 4.4: *Jane looks at David's information.*

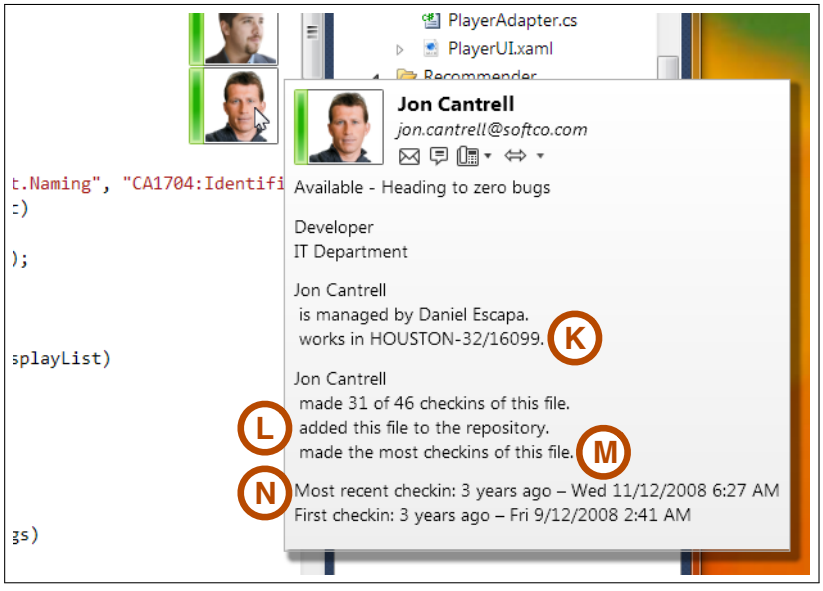


Figure 4.5: *Jane learns about Jon's contribution.*

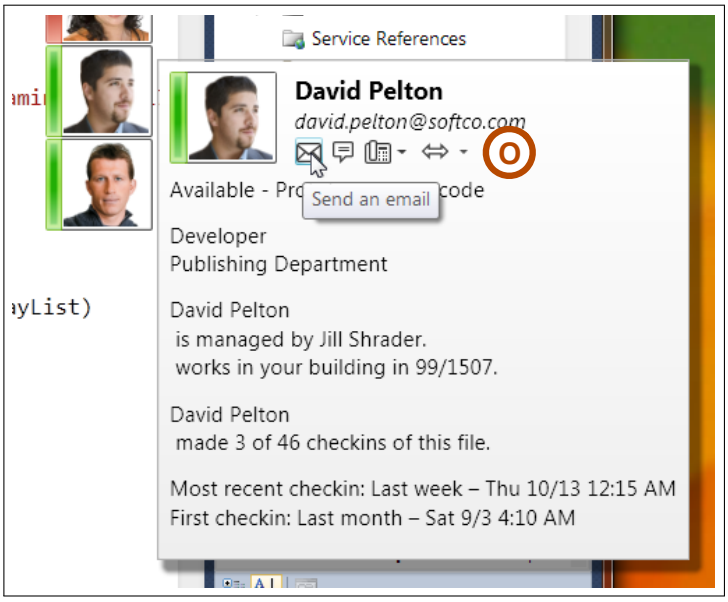


Figure 4.6: *Jane decides to contact David by email.*

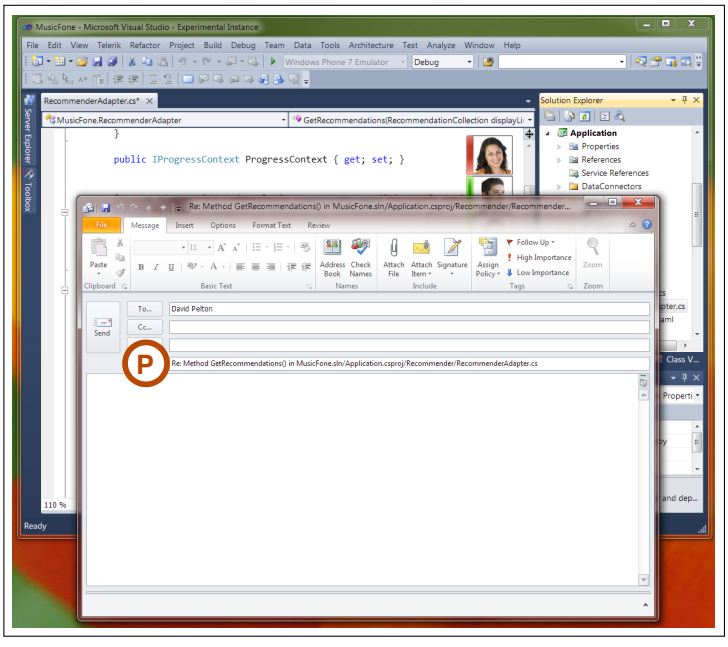


Figure 4.7: *CARES fills in the subject line of Jane's email.*

6. Since Phyllis is not available, and Jon is no longer an option, Jane decides that she will contact David (O in Figure 4.6). Though he is available, Jane feels uncomfortable sending an instant message to a new colleague. Instead, she writes him an email to introduce herself, and then asks her question.

   CARES helps Jane contextualize her email for David by filling in the current method and file name as the subject of the message (P in Figure 4.7). After a few minutes, David responds to Jane's email, and says that he is available for the next hour in his office. Jane heads over to meet David in person to learn more about the code.

By giving Jane the opportunity to read information about those who have worked on the source code, CARES made it possible for her to quickly discover and contact a relevant co-worker who can help her.

A video of this walk-through is available at `http://research.microsoft.com/~abegel/cares/demo.mp4`.

4.4.2 Tool Implementation

CARES is the first instance of a series of lightweight tools in the Codebook family of applications [13]. It is realized as a Visual Studio 2010 editor component extension. Its list of photos is a view-relative, embedded editor adornment (*i.e.,* a graphic effect layered on top of the text view that always remains on screen at the same position relative to the window borders), which shows as many photos as fits into the vertical space of the editor (though, during our deployment study, no one had files requiring more than 5 photos). To a first approximation, when a file from the source code repository is opened into the editor (detected using Visual Studio's built-in source code control extensibility object), a call is made in a background thread to the repository to fetch a list of all of the file's checkins. CARES collects the list of unique committers, and with each, associates the number and range of dates of their checkins, and fetches their company-managed, employee data (including their photo) from the repository. If the repository's committers also have Microsoft Windows identities, publically available extended information is pulled from the organization's Active Directory.

When the user hovers over a person's picture, a tooltip-like element (a Windows Presentation Foundation Popup with a tooltip-like visual appearance and behavior) is shown that contains individual and aggregate information about the person, their relationship to the file and the user, and optionally, their relationship with the other people being shown. When the Visual Studio user is running the Microsoft Lync communication software, each person's picture is augmented with availability information drawn from the Lync instant messenger status. Buttons in the Popup can then enable the user to initiate communication with the person via email, IM, phone, and screen sharing.

To ease its adoptability, CARES needs no configuration. It pulls required information from Visual Studio's built-in, pre-configured extensibility objects. CARES is robust to the loss

or recovery of connectivity to the source code repository and Lync server, gracefully degrading the user interface when needed data is unavailable.

The walk-through in Section 4.4.1 shows just one example of the information that CARES is capable of displaying. Its architecture employs the Managed Extensibility Framework to enable plugins, such as our source code repository plugin, to provide additional information about the current editor context. This context includes the current program element, file, project, solution, user, and any other associated people. Plugin authors can define alternative tooltip displays, such as showing number of changed lines per method instead of number of commits per file, using Windows Presentation Foundation XAML interface declaration files, without the need to alter the core source code of the tool.

4.4.3 Deployment Considerations

Many developers at Microsoft use Visual Studio, so it was a natural choice hosting the CARES extension. CARES uses Visual Studio's Managed Extensibility Framework, which enables plugins to easily extend Visual Studio internals (similar to what is possible with Eclipse) and allows plugins to be easily extended as well. CARES supports two kinds of extensions: source control repository access and employee metadata crawling. When a file is opened, CARES asks each source control extension to check if the file is managed by that source control system. If so, the extension asynchronously populates a list of email addresses and associated dates for each checkin and "shelveset" recorded for the file.[5]

From the CSCW literature and past experience deploying tools at Microsoft, requiring any configuration steps, server setup, or any type of waiting time between installation and use would hurt CARES' adoptability [69]. We took advantage of Visual Studio's built-in source control connections to obtain information about committers and their contributions to the code. We use Windows' connection to Active Directory (available at Microsoft and many other enterprises) to retrieve employee information. We piggyback onto the user's active Microsoft Lync unified communications session to provide our various communication modes. When the user is disconnected from the corporate network, or disconnected from Lync, CARES gracefully degrades the user interface by disabling the features that rely on those servers without affecting the others. For example, if the user's connection to the source code control server is severed, CARES cannot fetch the checkins and committers at all. If a new file is opened, CARES will display an error message to the user for five seconds and then fade out. If the connection comes back, CARES will reactivate and attempt to initialize itself again.

4.4.4 Understanding Identity

Identifying a person is much more complex than it first appears. To the source code repository, where CARES fetches information about checkins and shelvesets, the author is

5. A shelveset is like a checkin, but is intended only to be temporarily held while it is code reviewed by others.

an email address. In order to identify the name, contact info, and organizational information of the person represented by that email address, we simply have to find it in our Active Directory employee database. This may work, sometimes. But while the source code repository is recording past events, Active Directory only has records for **current** employees. If the email address cannot be found, it may indicate the employee has left Microsoft or changed his email address. Or, it might not be the email address for a human, but instead represents a no-longer-used machine account that made the checkin on behalf of an employee.

Consider this four-step example:

1. John Doe made a checkin in 1999 with the email address: `jdoe@microsoft.com`. He then left Microsoft for a startup in February 2000. Active Directory today has no record of a `jdoe@microsoft.com`, thus CARES would have no way to find out who `jdoe@microsoft.com` actually is.
2. In 2005, Jane Doe joined Microsoft and was assigned the unused email address jdoe@microsoft.com. When we now look up `jdoe@microsoft.com` in Active Directory, we will get Jane Doe's contact information. But, then CARES is misleading the user into thinking that Jane Doe contributed to the file six years before she joined Microsoft!
3. John Doe (the original) rejoins Microsoft in 2008 as a vendor. He receives the email address `v-jdoe@microsoft.com` — the "v" prefix identifies him as a vendor. An Active Directory lookup of `jdoe@microsoft.com` still returns Jane's contact info. CARES still believes the committer is Jane, even though John is a current employee.
4. Jane got married in 2011 and changed her name to Jane Public. To go with her name change, she changed her email address to `jpublic@microsoft.com`. Now, Active Directory will again have no record of `jdoe@microsoft.com`, thwarting CARES' lookup. Yet, John Doe does work at Microsoft, but is stuck with his vendor email address `v-jdoe@microsoft.com`. This leaves Microsoft with the right John Doe, but no way to link him to his former email address.

To address these scenarios, CARES uses the Codebook web service, which maintains a graph of software process information mined from the software repositories used by product teams at Microsoft. This information includes people, checkins, bugs, documents, tests, etc. Crucially for CARES, Codebook retains and makes available the entire history of all of the repositories. Every graph node in Codebook has the potential for recording a start date and end date when it was valid. Person nodes can store multiple sets of start and end dates, since people can leave and rejoin Microsoft many times. Each property of a graph node may be declared to be "revisable," *i.e.,* it too retains a record of every value it ever had along with the date range for which the property had that value.

Codebook mines its employee data from Human Resources, rather than Active Directory. Human Resources uses a much more complex employee database that contains historical information about current, former, and contingent (vendors, interns, contractors, etc.) employees. They distinguish employees not by name or email address, but by a personnel

number, which is unique to each individual and never reused. Even when a person leaves and rejoins Microsoft, they are assigned the same personnel number.

Unfortunately, the Human Resources database contains a lot of "dirty" data. Some data is missing when we expected it to be there (*e.g.,* the identity of the second author's manager is missing from the database for his first year of his employment). Some data is not applicable for a given date range (*e.g.,* a salesman working out of his home has no "office" phone number). And many date ranges themselves are inconsistent across different tables in the same database. We spent six months after building CARES learning how to clean the data into a coherent, consistent form. This cleaned data is what is stored in Codebook.

CARES invokes a Codebook web service API that provides the personnel number for any email address when also supplied with a single date. On any given day at Microsoft, only one person has a particular email address, so the combination of email address plus date is unique. CARES looks up each committer (or shelver) using the date of the checkin (or shelveset), and then retrieves (using another Codebook web service API) the complete historical record of that person's employment at Microsoft.

Codebook does not contain all of the information about people. Active Directory alone contains the person's photo and their instant messenger email address, both needed by CARES. Another employee database contains an important opt-out bit that indicates whether the employee wishes to make their photo public to people inside Microsoft. We must combine information from all three data sources in order to properly display the CARES UI. Codebook contains every historical email address for a person and their personnel number, while Active Directory contains every employee's current email address and personnel number. We can use Codebook's information to obtain the personnel number, and use that to look up the photo and IM address in Active Directory. We can then use the current employee email address to look up the opt-out bit in the third database.

Building such a pair-wise data lookup model made it difficult to design CARES with an extensible architecture that can incorporate additional employee information databases. However, if we could sufficiently modularize the employee lookup extensions, we could speed up CARES by parallelizing the lookups. Our solution is to use an AggregatePerson facade object which can dynamically aggregate and cache information from individual Person objects returned by each employee lookup extension. Each extension is requested to lookup an employee given a pair of an email address and a date. If found, the extension creates a Person object with whatever data it has available (including historical) and adds it into a CARES global Person Repository. Several of a person's properties can be used to uniquely identify them, either the personnel number, or a pair of a date range plus an email address, Exchange name (a person's name with qualifier, when necessary to distinguish the person from others with the same name), GUID, Windows Security ID, IM address. Whenever a Person is added into the Person Repository, its unique identifiers are intersected against those already in the repository. Whenever there is a match, those Person objects are aggregated into a single AggregatePerson facade.

For example, let us say that John Doe made a checkin. TFS identifies him as "`jdoe@microsoft.com @ 1999-04-20`." Active Directory identifies John Doe as "`v-jdoe@microsoft.com @ current`," as "`John Doe @ current`," and as "`Personnel Number 1234567`." Codebook identifies John Doe as "`jdoe@microsoft.com @ 1997-01-03 to 2001-02-10`," as "`v-jdoe@microsoft.com @ 2008-04-20 - current`," as "`Personnel Number 1234567`," "`John Doe @ 1997-01-03 to 2001-02-10`," and as "`John Doe @ 2008-04-20 - current`." The third database identifies John Doe as "`v-jdoe@microsoft.com opt-in @ current`." The Active Directory Person and Codebook Person overlap in the personnel number, so they are merged together. The email address of the third database's John Doe overlaps with the email address of the Active Directory Person, allowing it to be merged into the AggregatePerson object as well. Finally, the email address and date of the TFS Person overlaps with the date range of the Codebook person's first email address. This last match lets CARES associate its checkin with the John Doe AggregatePerson object and display his correct, current contact and organizational information in its user interface.

So, what if the person who made the checkin is found in Active Directory but not in Codebook? This can happen when employee checkins are gated by quality testing — all employee checkins in the same time period (e.g. hour) are grouped together and tested. If the tests succeed, a machine account then commits all of the checkins to the repository. Machine accounts are Windows principals, and thus exist in Active Directory, but since they are not employees, they do not exist in the Human Resources database that supplies data to Codebook. We noticed that a secondary way to confirm the email address belongs to a machine account is that its Active Directory record contains no manager.

Sometimes an email address from a checkin cannot be found. If the Human Resources employee database were infallible, we would be able to conclude it is a machine account that is no longer used. However, the HR database has some missing rows in its email address to personnel number table, preventing us from linking his contact information to his email address. To reduce the incidence of these "missing" employees, we have manually curated 1,500 former employee's table entries, using manual inspection. The names and email addresses are often very similar to one another making them easy for a person to identify.

4.5 Evaluation

We deployed CARES twice, first in a pilot to 30 developers, and then in a Microsoft-wide internal release. Until the end of June 2012, it has been used by 106 additional employees. The reaction to CARES by most individual contributor developers has been primarily positive, however, a few people felt that it did not fit with the ways they preferred to communicate with others. In this section, we describe both the positive and negative reactions to CARES, illustrating the diversity of communication styles used by employees sharing the same role.

Eight pilot users were interviewed. All of them liked the CARES tool itself. "CARES *is pretty cool*" and "*awesome.*" One developer liked it enough to show it to his manager, who said, "[it] *puts a face to the code. Now I know who to talk to.*" That manager went on to ask his entire team to start using CARES. All of the Survey 3 respondents said it was clear why the people who showed up in CARES were there (100% N=23).

They told us that CARES simplified and sped up their process for finding relevant engineers. One interviewee said, I "*would use the* CARES *tool to get their name, email and contact card with their office address.* [It] *saves me time from running* [a code history tool] *and* [an address book tool]." Another said "*The more I can just stay here* [in the IDE] *where I'm doing my work, the better it is.*" A third said he "*looks at* [CARES'] *availability indicator to confirm* [that someone is] *free*" "*before walking over. Green: yes, Red: message, Yellow: wait or email.*" 56% (N=18) of Survey 3 respondents reported that seeing the person's availability indicator helped them to decide who to contact at that moment.

Developers spoke of situations where CARES had helped them. One said, "*The add-in is helpful for me... There's lots of people who have implemented* [code] *in the past, and I have to understand them all.*" Another said it was "*handy to know who worked on that particular code, especially when it was developed by someone on a different team years and years ago.*" Two others predicted that people who used CARES would end up asking *them* questions, one because he worked on the product's core which everyone else used, and the other because he was in the same team for many years, and had contributed to almost every file.

In the CARES design, we chose to show photos instead of names because we wanted to *encourage* communication. The photos are always visible, giving the developer the feeling that someone out there *cares* about his work, and is keen to be contacted about it. The CSCW literature backs up our intuition, suggesting that visual cues provided by photos can help people identify individuals' relevant social categories and promote shared social identity among colleagues [142]. 48% (N=21) of Survey 3 respondents strongly agreed or agreed with the statement, "*Seeing the faces of the contributors to the code helps me to feel like I am part of a community.*" 43% were neutral and only 1 respondent (5%) disagreed.

CARES' photos enable group members to easily recognize one another, which increases their sense of belonging. One interviewee said it was "*definitely easier to figure out who the people are with the pictures.*" Another showed us how he investigated pointers to Visual Studio solutions that he received in his email. Having never looked at the solution's code before, and then opening it into his CARES-enabled IDE, he exclaimed, "*that's* [name omitted]*! He's actually on my new team. It'd be real easy for me to talk to him.* [Pause] *That would have definitely taken me longer without* CARES*. I would be trying to hunt people down.*" His feeling was shared by many in Deployment 2. 57% (N=21) of Survey 3 respondents strongly agreed or agreed with the statement "*For those people whom I had met before, seeing their faces in* CARES *helped me to recognize them.*"

Not everyone found that CARES was the right tool for them or their team. While 81% (N=21) of Survey 3 respondents reported that CARES showed enough information to understand the recency of a person's contribution to the file, only 39% (N=23) felt there was

enough information to see the *magnitude* of a person's contribution to the file. 5 Survey 3 respondents requested that CARES show line-by-line attribution each file, rather than aggregate all contributors to the file. We have been reluctant to add this feature since it would duplicate the functionality of the "annotate" function (*i.e.,* blame) in Visual Studio with TFS. Showing relevant people per line also presents a user interface challenge because you do not want your tool to attract the user's attention with UI changes when it is irrelevant to the user's main task. [39].

In our interviews, we spoke with one developer who said that while it was useful to know which developer in India wrote some code, he still preferred to contact the developer's team *liaison* in India to ask questions. He explained that his way enabled the liaison to delegate his question to anyone who was relevant and available to answer the question. With our tool, if the chosen developer was unavailable, or out of office, the answer would have been delayed at least 24 hours.

Another developer, the most senior member of his team, said that people came to him to find out who to talk to. "*I could gauge whether their question was appropriate, or in my estimation, they should have done more homework.*" He worried that people on his team would use CARES to speak directly to more relevant colleagues without "*demonstrating that they had any mental model at all.*" He noted further that CARES was of no use to him because "*it's almost always clear who the author is, or who the right person is to contact... I know, based on my years of experience... who the author is.*"

In Deployment 2, we employed logging to record developers' interactions with CARES. We have processed basic usage data (shown in Figure 4.8). In its second deployment, CARES has been used by 106 software engineers all across Microsoft (not including Deployment 1 users). More importantly, CARES was not just used once or twice, but continues to be used on a regular basis by 36 of those engineers. Most small utilities deployed in an enterprise lose users quickly, due to serious bugs, overcomplexity, or user boredom; that CARES has remained installed and used by developers bodes well for its future.

4.6 Related Work

Coordination studies have a long history in software engineering and CSCW research. Sarma [152] presents a comprehensive review of coordination tools, with features ranging from *live team awareness*, to *top-down socio-technical exploration (STC)*, to *IDE-based tools*, to *context-sensitive tools*. Examples of such tools are:

- **Live team awareness:** Palantir, TUKAN [156], FastDash, Jazz, and CollabVS;
- **Top-down socio-technical exploration (STC):** Expertise Browser, Palantir, FastDash, Ariadne and Tesseract (both use Cataldo's STC), BeeHive, and Codebook;
- **IDE-based tools:** IBM Rational Team Concert (née Jazz), CollabVS, Palantir, TUKAN, Deep Intellisense, and TeamTracks;
- **Context-sensitive tools:** TUKAN, TeamTracks, and Deep Intellisense.

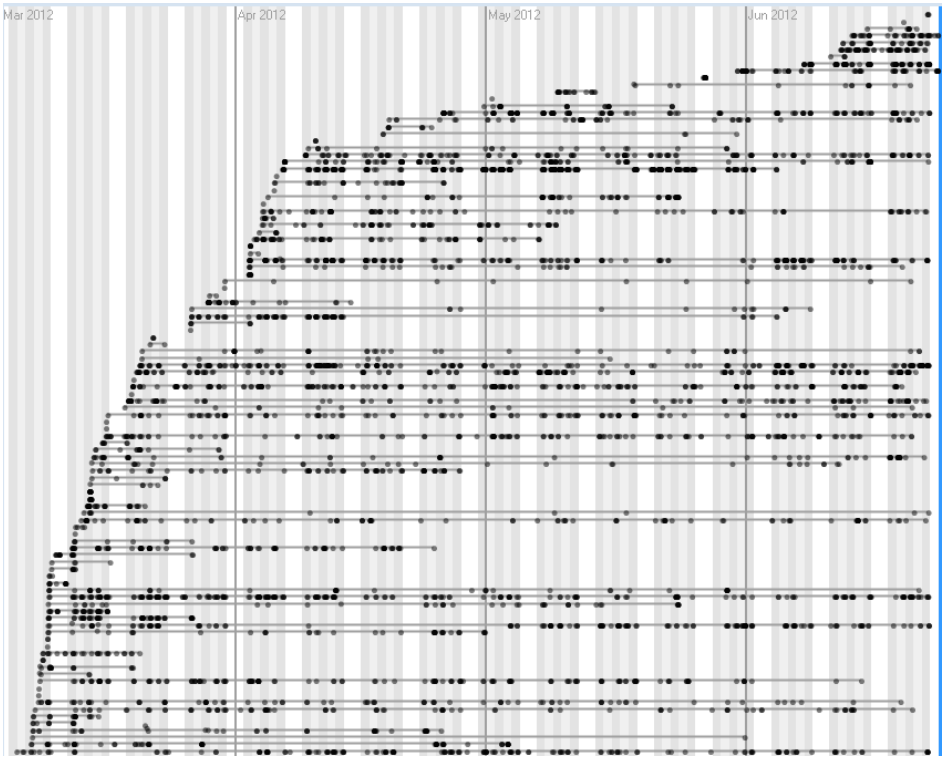


Figure 4.8: *Usage logs of CARES users. The X axis is the date and the Y axis shows each unique user. A dot on a row shows when the person used CARES in his or her Visual Studio session. The timeline begins on March 8, 2012 and ends on June 24, 2012.*

Unlike the live awareness tools, CARES aims at developers on large enterprise software teams working independently on components of a software product [103, 110, 49, 44, 35]. Each developer may care little of the daily work of people on other teams (if they know them at all), except when a coordination request or information need arises. As in Costa *et al.*'s study [35], the products we studied are also long-lived; relevant contributors may have moved on to unrelated projects. This reduces the need to see others' current activities and allows a simpler display of availability in line with the literature's recommendations [39]. While some of CARES' design elements overlap with Schummer's awareness design patterns [156], other aspects harmonize with Nakakoji's guidelines for communication with experts: being personalized, contextualized, and socially aware (though we do not limit users' communication modes) [125].

CARES is context-sensitive and contextualized to the IDE, only showing the people relevant to the user's focus. This design (also used by TUKAN and Deep Intellisense [90]) requires a simple glance and mouse hover to choose with whom to communicate. The

top-down STC tools require search, browse, or information-pivoting operations to move from a global view to the desired context. Jazz, CollabVS, and Palantir always show everyone on the team, while TeamTracks anonymizes the people and displays their aggregated IDE actions. Only Ariadne and CARES show how people are related to one another and to the tool user. Ariadne shows a similarity-metric-based person graph, while CARES shows concrete relations, such as relative office and organizational location, and relative code contributions.

There are many possible methods to select relevant people to show (e.g. organizational charts, program analysis, whole-system STC graph analysis [29], email [91], degree of interest functions [101, 61], and newsfeeds [58]). Our CARES prototype uses committers as a simply-computed, ecologically valid proxy for ownership and knowledge. This choice drastically improves CARES' deployability by avoiding the need for offline analysis and custom servers (which are required by many of the other tools). CARES is auto-configured from the project's source control context and works immediately after installation.

While many coordination studies have looked at why software engineers communicate and then pondered the implications of too little or too much communication, ours is the first to discover the criteria engineers use to select the right person to speak with. In addition, we noticed that very little of the literature describing communication and coordination tools offers longitudinal data about usage, whereas our report describes two deployments over a total of five months of tool use at a large software enterprise.

4.7 Concluding Remarks

Enterprise software development is notable for supporting large numbers of engineers working for long periods of time on projects that have significant amounts of legacy code. At Microsoft, we noticed that ad hoc, asynchronous, intermittent communication between software developers about the source code was common, yet poorly supported by the general communication tools in daily use: email and IM. We found, after several months of study, the methods and "algorithms" that software engineers use to discover and select relevant people to speak with about their code. We learned that although their communication was intermittent, it was a key factor in establishing long-lasting work relationships that help make future collaborations less difficult.

By building an IDE-based tool to specifically support the person discovery and selection process, we realized that modeling employee identification in a large long-lived enterprise was very complex, and was often made so by the multitude of complementary, yet inconsistent employee metadata databases maintained by various corporate departments. Correlating information and dates across the databases was essential to uniquely identifying individuals who had contributed to source code repositories but had since changed their organizational affiliation or personally identifying information (PII). Our techniques can be used to efficiently disambiguate or "unify" individuals across many different kinds of PII metadatabases.

After distributing our tool twice (once for three weeks and once for four months) we confirmed that deployability is strongly influenced by ease of installation, simplicity of use, and effectiveness at a single task. We plan to continue studying the impact CARES has on developer to developer communication at Microsoft. Our long-term goal is to learn about and support communication scenarios between developers and non-developers. With communication comes cooperation and trust, and with both comes more effective and successful collaboration.

5 Bellevue: Receiving Changes

Teamwork in software engineering is time-consuming and problematic. In this chapter, we explore how to better support developers' collaboration in teamwork, focusing on the software implementation phase happening in the Integrated Development Environment (IDE), where developers spend most of their time.

Based on one of the recommendations presented in Chapter 2 we analyze the current IDE support for receiving code changes. We find that historical information is neither visible nor easily accessible. Consequently, we devise and qualitatively evaluate Bellevue*, an IDE extension that makes received changes always visible and code history accessible in the editor.*[1]

1. This chapter contains the second and last part of the paper "Supporting Coordination in the IDE" [74], accepted for publication in the proceedings of the 18th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2015). The authors of this publication are Guzzi, Bacchelli, Riche, and van Deursen. This paper has been awarded a Best Paper award.

5.1 Overview

Based on one of the recommendations of our exploratory investigation presented in Chapter 2 (Section 2.4.3), in this chapter, we investigate how to improve the current approach of receiving changes in the IDE, because currently support is minimal and not well integrated in the IDE.

We first briefly recall the current state-of-the-practice to receiving code changes. When developers decide to receive new changes in the IDE (regardless of the IDE or the versioning system used), they receive the complete list of changed files. Subsequently, they can inspect each file with a read-only view, and visualize the textual differences from their current version. When developers eventually deem the changes are appropriate, they can (semi-)automatically integrate them (*i.e.*, merge) with their local copies. After the merge, the context of receiving changes is closed, and developers continue working on the updated code. Figure 5.1 exemplifies a typical user interface employed during this process.

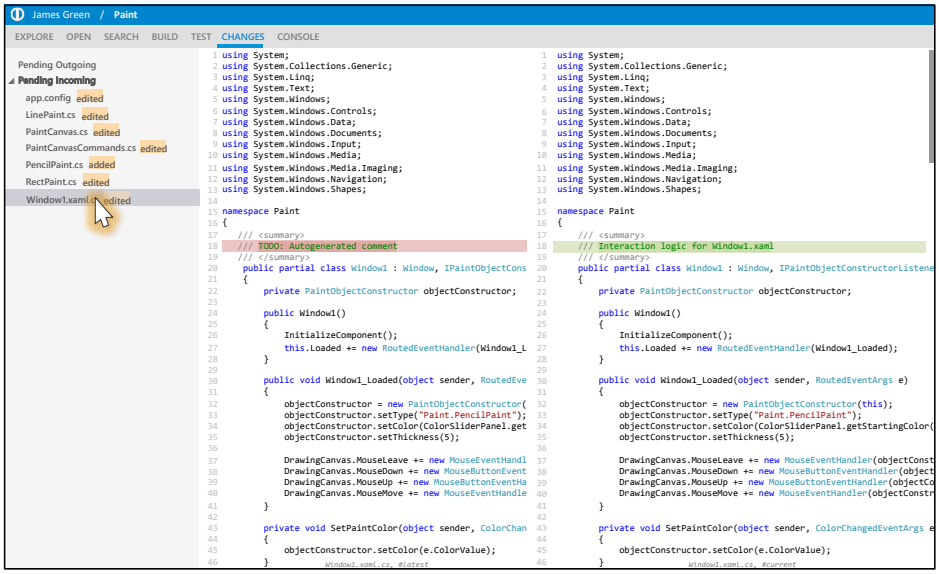


Figure 5.1: *Exemplification of current UIs for receiving code changes*

By analyzing the support for receiving changes in the IDE under the light of widespread usability heuristics [122, 127], we find that it does not properly support the information needs of developers about change history. Based on this and on the findings from the exploratory investigation we (1) derive requirements for an IDE extension, (2) design Bellevue, the design of a prototype fulfilling these requirements, and (3) iteratively evaluate and improve Bellevue with 10 senior industrial developers from different companies and with different backgrounds.

Bellevue makes newly integrated code changes always visible, both at project and file level, and code chunks' history easily accessible. It shows historical information within the active editor to allow users to modify the code without switching context. Bellevue takes historical change information that is already available and offers an interactive view that shows detailed historical information for files and specific chunks with respect to a previous version. In this chapter we present the Bellevue tool design and evaluation.

5.2 Methodology

The goal of this chapter is to investigate a technique to improve teamwork in the IDE, by following the recommendations discussed in Chapter 2. In particular, we focus on receiving changes in the IDE and how to improve its support. In the following we detail our approach (illustrated in Figure 5.2) by dividing it in the two main methodological steps we pursued: design prototyping and RITE-based design evaluation.

5.2.1 Design Prototyping

As a first step, we analyzed the current approach for receiving changes in the IDE under the light of widespread usability heuristics [127] (Point 1 in Figure 5.2). We found several unmet heuristics that, together with the data collected in the exploratory investigation, we used as a basis to derive requirements for our IDE extension to improve receiving changes and support teamwork (Point 2).

Afterwards, we devised an IDE extension, named Bellevue, to fulfill the requirements. To explore our preliminary design ideas for Bellevue, we employed *concept prototyping* [112]: We created initial concept sketches of Bellevue on paper (Point 3), which we used for communicating our ideas to various industrial user experience (UX) experts (Point 4). This let us to get their feedback, reveal early problems, and improve the initial concept.

Once the concept of Bellevue was more solid, we devised a detailed storyboard including a high-fidelity prototype of Bellevue (Point 5). This was implemented in the form of a slide deck, a technique commonly used in professional UX prototyping, which contained a sequence of believable action steps of interaction with the prototype. Each step was devised in a way that the participants of the evaluation phase could observe what was happening, explain what they would do, and describe the effects they would expect as a consequence of their actions.

Compared to implementing a full-fledged version of Bellevue to evaluate with users, this prototype based approach allowed a faster feedback cycle to improve the design and to incorporate comments from the users during the evaluation.

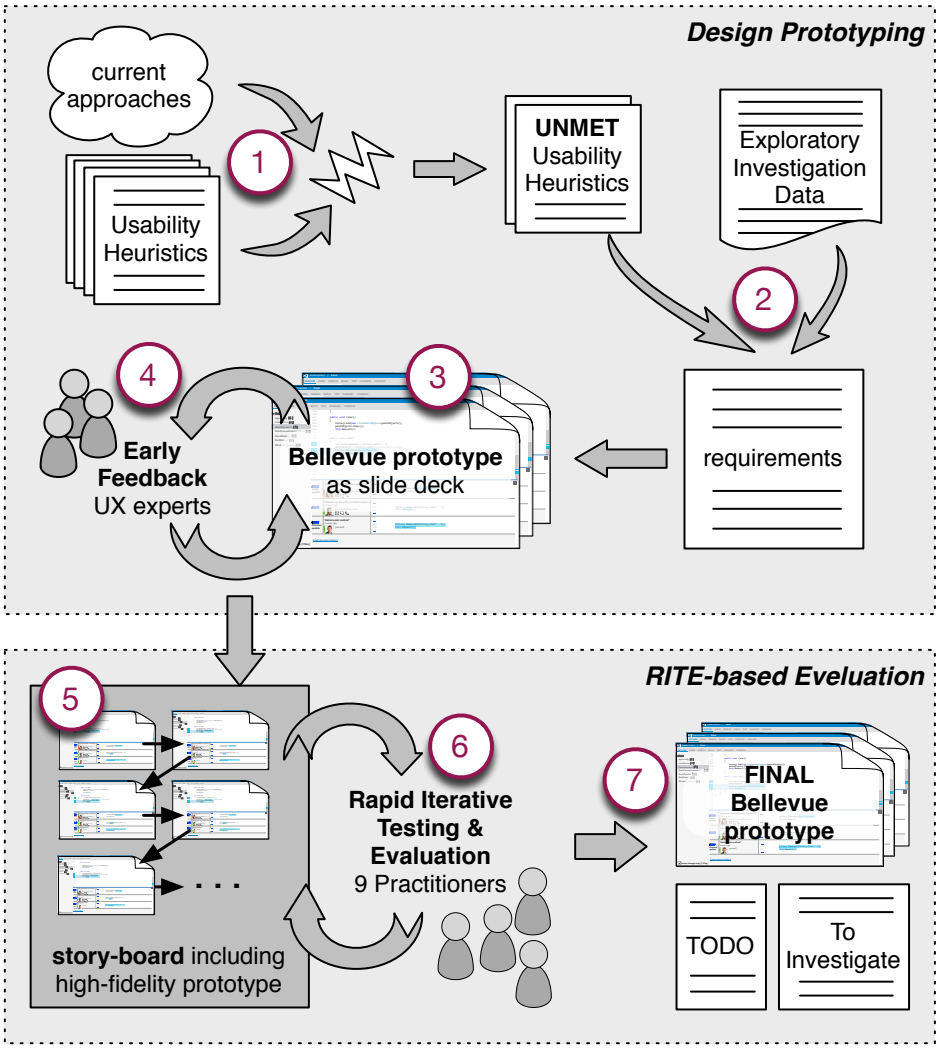


Figure 5.2: *Overview of the applied research method*

5.2.2 RITE-based Design Evaluation

We evaluated the high-fidelity prototype of Bellevue with professional software developers. We conducted the evaluation using the RITE (Rapid Iterative Testing & Evaluation) method: A formative usability testing method used to gain exploratory insights into the user behavior with the intention of quickly iterating and fixing problems [119] (Point 6 in Figure 5.2). RITE allowed us to evaluate and identify problems in the prototype, quickly fix them, and then empirically verify the efficacy of the fixes, using a rapid test-fix-test-fix cycle.

We planned to have at least 8 iterations, each one with a different professional developer. If problems were still present in the design after these cycles, we could extend the number of iterations, until we reached a stable version. Participants were selected among a population with the following characteristics: More than three years as a professional developer, more than one year in the current company, and more than three months in the current team. Moreover, from a more technical perspective, interviewees had to spend at least 20 hours per week on writing and editing code, their team had to use a source control management system, and they had to have at least browsed the change history, encountered a change merge, or used the file diff comparison view in the month before the RITE. Evaluation invitees were encouraged to participate by a gratuity in the form of software offered by Microsoft.

Each participant was invited to come to a usability laboratory at Microsoft, where we conducted this study. We informed each of them that the session would have been video-recorded. To minimize invasiveness and the Hawthorne effect [135], only one researcher participated and guided the session, and the video-recording was done through hidden cameras. There were three cameras from different angles to evaluate: (1) the reaction of the participant to the presented design, (2) the interaction with the screen, and (3) the communication with the researcher guiding the session. To mitigate the *moderator acceptance bias* [63], we explained each participant that the product was not created by the researcher guiding the session, but by an external team. Moreover, to mitigate any *social desirability bias* [63], the storyboard plot was describing the actions taken by a proxy developer named James. This also encouraged developers to comment and discuss their thoughts without the fear of suggesting something that would be considered "wrong" when it would not eventually happen in the plot.

Following the storyboard plot described by the slides and the researcher, participants were solicited to follow a think-aloud protocol, and indicate what they saw, would do, and would expect as a result of their actions on each screen page. The role of the researcher was mostly to encourage the participants to think aloud, to confirm what the users suggested as actions showing the following slide, or to motivate the subsequent choices shown in the storyboard, if different from users' expectations.

Both by taking notes during the session, and by analyzing the recorded videos, we documented areas of difficulty and positive reactions. In particular, we kept track of features that users suggested (implicitly or explicitly) to change, and of aspects that could be fur-

ther investigated. When a participant underlined a clear error in the prototype or when there was enough data to justify a change, we discussed it and we changed the prototype before the next iteration to test it with the next participants.

Eventually, after 9 iterations we reached a stable and validated design. At the end of the process (Point 7 in Figure 5.2), we had: (1) the finalized Bellevue prototype, (2) a set of changes to implement but that were not eventually integrated, and (3) a set of candidate aspects to be investigated as future work.

5.3 Tool Requirements

Although the current approach (exemplified in Figure 5.1) to receiving code changes is widespread, our exploratory investigation (presented in Chapter 2) let emerge that it is not optimal and offers developers minimal support. To pinpoint actionable problems that could be solved by a tool, we analyzed the current approach under the light of the broadly accepted [112] usability heuristics presented by Molich and Nielsen [122], and further refined by Nielsen [127]. This analysis let emerge a series of problems (in line with the findings of our exploratory investigation): Many widespread usability heuristics are not met by current support for receiving changes. Satisfying these heuristics defines the requirements for Bellevue, our tool to provide better support for receiving changes in the IDE. In the following we describe the unmet usability heuristics. We quote the developers interviewed in our previous exploratory investigation, referring to them as D1-D11 (see Table 2.1).

5.3.1 Recognition over Recall

> "Memory for recognizing things is better than memory for recalling things." [112]

Thus, interfaces should not force users to recall information from one part of an application to another.

This usability heuristic is not met by development environments: Once the developer decides to merge a received change with the local version, the information about the integrated change is not visible anymore. For this reason, when developers encounter a bug, they must recall which files changed and whether they could have generated the problem. One interviewed developer explained that the frustration when he encounters a bug comes from "*figuring out where the problem is: Trying to figure out what really has changed*" [D5]. When looking for the cause of a bug, developers memory can be aided by tools to navigate change history, but such tools require to switch from the current development context, and they only gives information concerning the changes without the current development context.

5.3.2 Visibility of System Status

> "The system should always keep users informed about what is going on." [127]

This is not met by change management: Once changes are integrated, there is no distinction between the lines already present before the merging, and the newly integrated ones. Therefore, there is no clear visibility of the system status with respect to its history. On this topic, one interviewed developer explained: "*It's kind of impossible to know every single line of code that everybody else on your team changed*" [D3]. In fact, historical information is available, but only in dedicated tools/views out of the current development context, thus the status is neither self-evident nor easily accessible: "*there isn't really an easy method [...] that let you see [that] these ten files are different from what you had in your current time*" [D5].

5.3.3 Clearly Marked Exits

> "A system should never capture users in situations that have no visible escape." [122]

From the code change management perspective, developers have an escape: If they find something not working after they merged some changes into their local working copy, they can roll back to the status prior to the merging. There are two problems with this approach: (1) the exits are not evident and (2) the exit strategy is binary.

Due to the former problem, developers have to realize themselves (by recalling the merged changes) that one way to solve the problem at hand could be to undo the merging, instead of trying to find an error in their own code. Due to the latter, it is necessary to undo *all* the merged changes at once, although the error can be caused by a mistake in a small fraction of the changed code. Once the code is rolled back, developers have to reconsider all the undone changes and realize which ones could have caused the error, without having the full IDE context at disposal, but only the change information, and integrate all the unrelated changes back again. D1 explained: "*It's a loss of time, we have to roll back, figure out* [what the problem was]*, and roll again. It's a loss of time, definitely.*"

5.3.4 Help and Documentation

> "[Documentation should] be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large." [127]

Concerning code changes, the documentation often consists in the commit message that the author of the change wrote to explain it. This is visible in the interface used to inspect received code changes (*e.g.,* in Figure 5.1), but once the changes are integrated it disappears, unless the user performs a number of steps navigating the history of the code within specialized windows or applications. For example D5 complained: "*When you get the latest* [changed files] *you get tons of files*"; he found it very difficult to search the necessary help or information due to information overload. Finally, when developers integrate more than one commit into their local copy, often they see only the last commit message,

even though a line of code could have been changed several times between their local copy and the current status.

5.3.5 Help Users Recognize, Diagnose, and Recover from Errors

Current code change management in the IDE makes it difficult to recognize and diagnose errors generated by integrated code changes, because they are not visible and their history has to be analyzed outside of the current development context. One interviewed developer explained that, despite the availability of external history tools, "*one of the problems is trying to figure out what really has changed* [and] *what's the impact on your code*" [D5]. In fact, as D3 explained, external tools are not helpful because "*version control gives you a list of files that changed and not the specific lines*": Seeing exactly which part changed and how takes many steps. Moreover, currently the only possibility to recover from errors is to completely undo the merged changes, while perhaps it would be sufficient to modify a small part of code to remove the error.

5.4 Design Features and Evaluation

We designed Bellevue to make received code changes always visible, to favor recognition over recall, both at project and file level. We made the history of code chunks easily accessible using progressive disclosure, to help users recognize and diagnose errors. We also made it possible reverting from specific code changes to provide users with marked exits to recover from errors. Bellevue shows historical information within the active editor to allow users to modify the code without switching context.

Bellevue aims at being a lightweight solution, ready to be used, without requiring developers to change their working style. It takes the historical change information that is already available, but currently neither visible nor easily accessible, and displays it in a non-obtrusive way. Bellevue offers an interactive view that shows detailed historical information for files and specific chucks with respect to a previous version.

We detail the features of the prototype, as they were at the end of the RITE phase, and the corresponding feedback from participants, including future topics to investigate. As previously mentioned (see "RITE-based Design Evaluation"), nine developers participated in the evaluation (we refer to them as R1–9). The final version of the slide-deck used in the RITE phase is available in the online appendix [71].

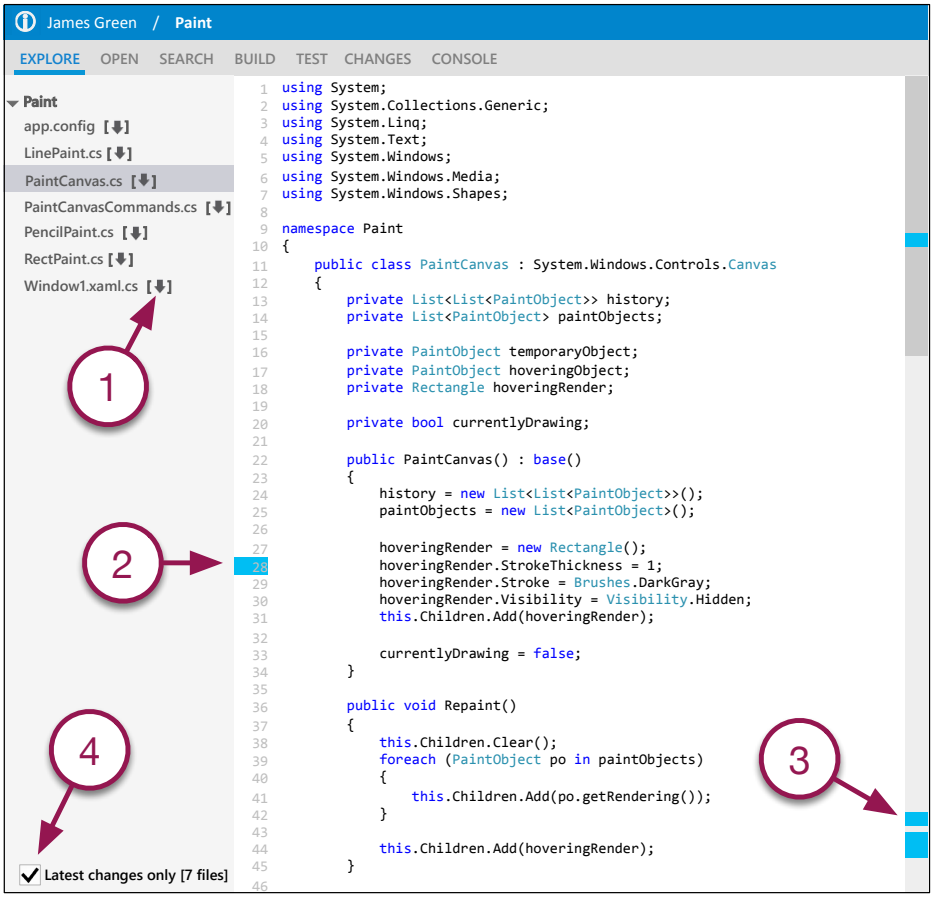


Figure 5.3: *Recognizable changed files and blocks, and filtering*

5.4.1 Recognizable Changed Files and Blocks

Bellevue decorates changed files with an arrow (Figure 5.3, Point 1), and denotes changed lines with a blue[2] colored sign, both at a fine-grained granularity (Point 2), to see them in the context of the current text window, and a more coarse-grained one (Point 3), to see them in the context of the entire file. One can decide (Point 4) to see only the files that were just merged into the current local version. This design supports recognition over recall: Once new changes are merged into the local version, their traces remain visible. It also enhances the visibility of the system status, with respect to changes.

2. The color blue has been chosen because it is currently considered a neutral color in IDEs, as opposed to green, orange, or red, which are often associated with versioning systems or debuggers.

RITE participants' feedback—All the participants appreciated this feature. In particular, they liked that it helps filtering out irrelevant information when looking for the reason of an error that could have been introduced by a received change: "*Knowing what I can ignore is huge, the larger the project, the more beneficial it comes*" [R1]. Concerning the way in which changes are made recognizable, some users did not find it intuitive, or appropriate: "*I'd prefer a bar or something much more visibile* [than a blue-colored sign] *to see that it's different*" [R2]. Nevertheless, after they continued in the scenario and experienced the following features of Bellevue, they withdrew their concerns. Some participants suggested to let the users personalize the color to denote changes; other participants suggested to use different colors to clearly distinguish added, removed, or modified lines, as it currently happens in tools that display code differences.

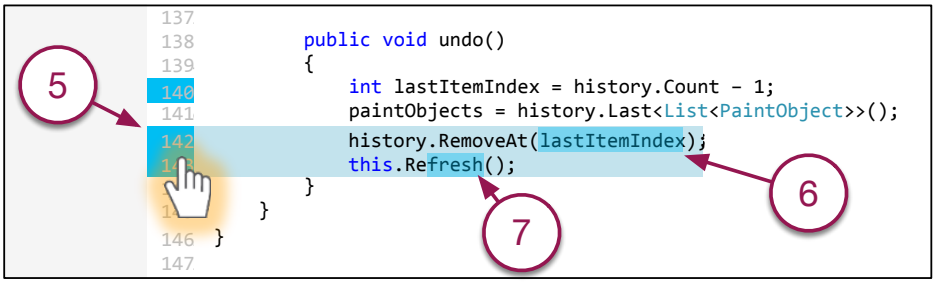


Figure 5.4: *Visibility of changes' effect by mouse hovering*

5.4.2 Visible Changes' Effect

To show the effect of the change in the code, the user can hover on any colored block to see the latest changes. For example, in Figure 5.4, the user decided to look at the changed block that was not visible in Figure 5.3. Then, by hovering on the colored sign on the left (Point 5), (s)he can see the effect of that change: The argument of the `RemoveAt` method call has changed (Point 6), and the `Refresh` method call has replaced a call present before on the same object (Point 7).

RITE participants' feedback—This feature was introduced in the third iteration of the tool, after considering the feedback received by the first participants. As an example, one participant had some expectations when hovering the lines indicating a change: "*toggle to highlight what's different from the last version, to quickly diagnose, I don't need a true side by side*" [R3]. Once introduced, this feature was well received by all the remaining participants (*e.g.*, "*ok, good! I can see here how* [this part] *changed!*" [R6]), because it also helps with the *progressive disclosure* [97] of the information about the changes: Users can quickly verify whether the changes seem relevant and, only if necessary, investigate more.

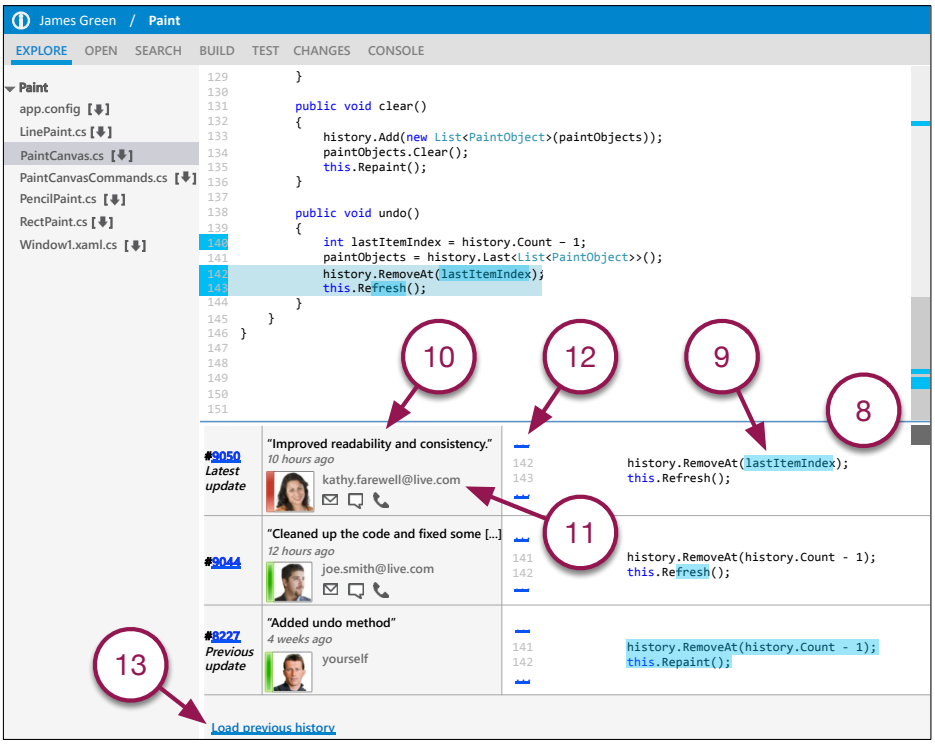


Figure 5.5: *Accessible historical details*

5.4.3 Accessible Historical Details

In Bellevue the user can see the code history of any block that was changed with respect to the previous local version. This is achieved with one click on the colored sign on the left of the interesting block. For example, in Figure 5.5, the user decided to further inspect the history of lines 142–143 because they led to an unexpected behavior. Once the block is clicked, a pane appears from the bottom (Point 8): It contains the historical details of the changes happened to that block since the last update of the user. Each item represents a change and shows the changed code with respect to the previous commit (Point 9), the commit message to document it (Point 10), and the contact information of the change author (Point 11). The displayed changed code only regards the chosen block, but it is possible to extend it by clicking on the '...' before and after the code lines (Point 12). Previous history can also be inspected (Point 13).

RITE participants' feedback—As for the other steps, before showing what would happen next, the interviewer asked the participants how they would interact with the design and what their expectations would be. In particular, for this feature, the interviewer asked

what participants expected it would happen by clicking on the colored sign on the left (Point 5). In this way, we learned that the participants wanted to have something similar to a diff with the previous version (*e.g.*, "*I'd do a compare versus the previous version, and just look at those particular changes*" [R3]). The Bellevue solution was, thus, very much appreciated and it often exceeded their expectations: "*All the details! This is exactly what I was looking for: It tells me who* [...] *and it tells me what did each one, and how long ago!*" [R1]; "*oh I see, so this is exactly what I was looking for. It's even better!*" [R8]. Seeing the version that could have introduced the error (*i.e.*, #9044) was a clearly marked exit: Some participants considered the possibility of recovering the error by reverting that particular change, because that would not imply reverting entirely to a more complex change set.

Over the different iterations, we added the clickable revision number (to open a panel to see all the changes in a revision), and the hovering function over a commit comment to show the full comment.

Participants' suggestions that we did not eventually include in the iterative evaluation, due to time reasons, mostly regarded the possibility of selecting specific parts to see the history, instead of the contiguous block (*e.g.*, "*I want to see the whole function* [history, by] *right clicking on a function*" [R2]).

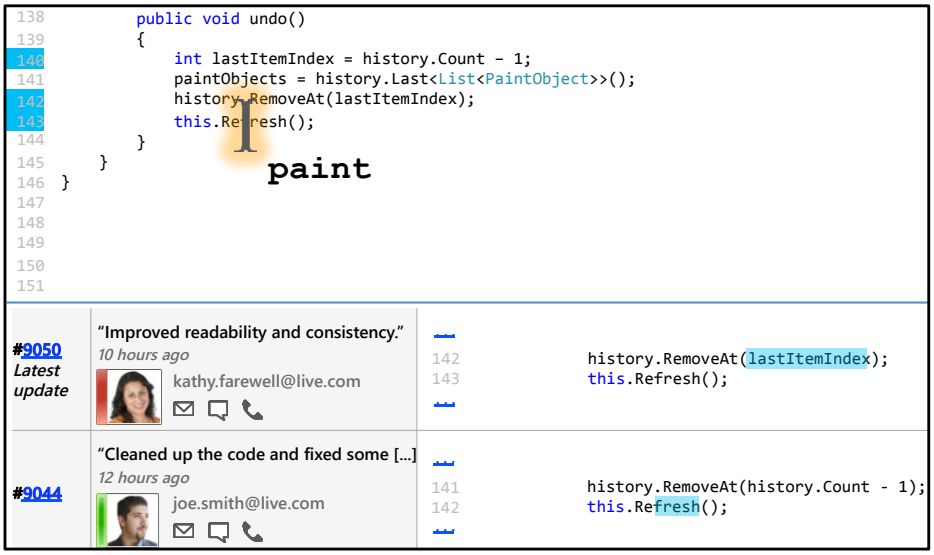


Figure 5.6: *Editable code while accessing historical details*

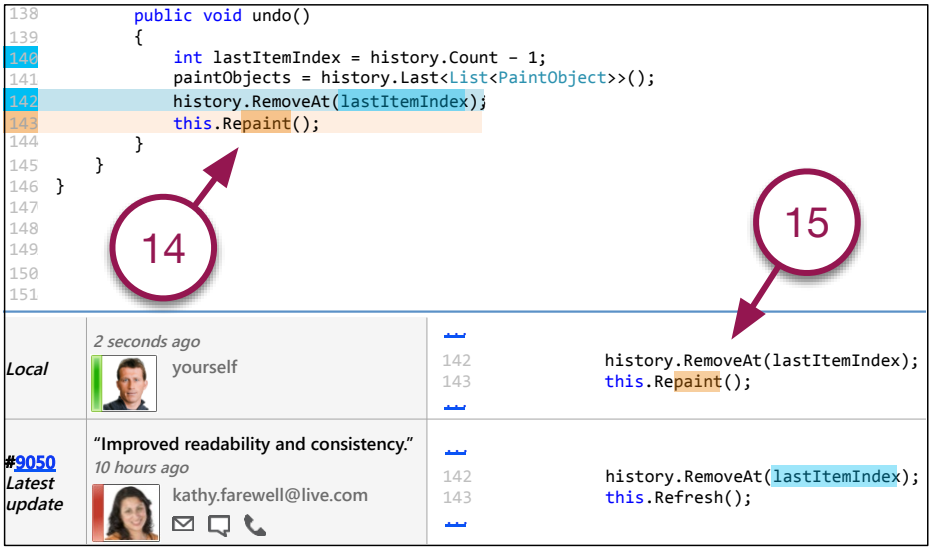


Figure 5.7: *New local change added to history*

5.4.4 Editable Code

Bellevue allows developers to edit code while reviewing the history (Figure 5.6), because it integrates history within the active editing context. It also highlights the new code differently (Point 14 in Figure 5.7) and automatically adds a new item in the list (Point 15) to put it in its historical context. This differs from current approaches for visualizing history, which involve opening a distinct context in the IDE or a completely different application, and do not make it possible to edit code (*e.g.*, to fix a bug) and see history in the same context, at the same time.

RITE participants' feedback—This feature was also very well received by all the participants. In particular, a number of them was positively surprised and realized the possibilities of having code changes better integrated in the IDE: "*I have a diff view, but I am not trapped in that* [...] *I got my editor and my diff view, so the split view is very very helpful* [...]. *Let me do what I want to do, while looking at the information I needed to make my change*" [R1]; "*Now that I see, I know what is happening* [...]. *That is intuitive to me: Just clicking, edit, and go*" [R7]. They also appreciated the immediate feedback of the change in the local history (Figure 5.7): "*Oh, I like it shows it's local*" [R4].

5.4.5 Contacting Change's Author

The author's photo and contact pane is inspired by cares (Chapter 4), our tool to help developers discover and choose relevant colleagues to speak with when seeking informa-

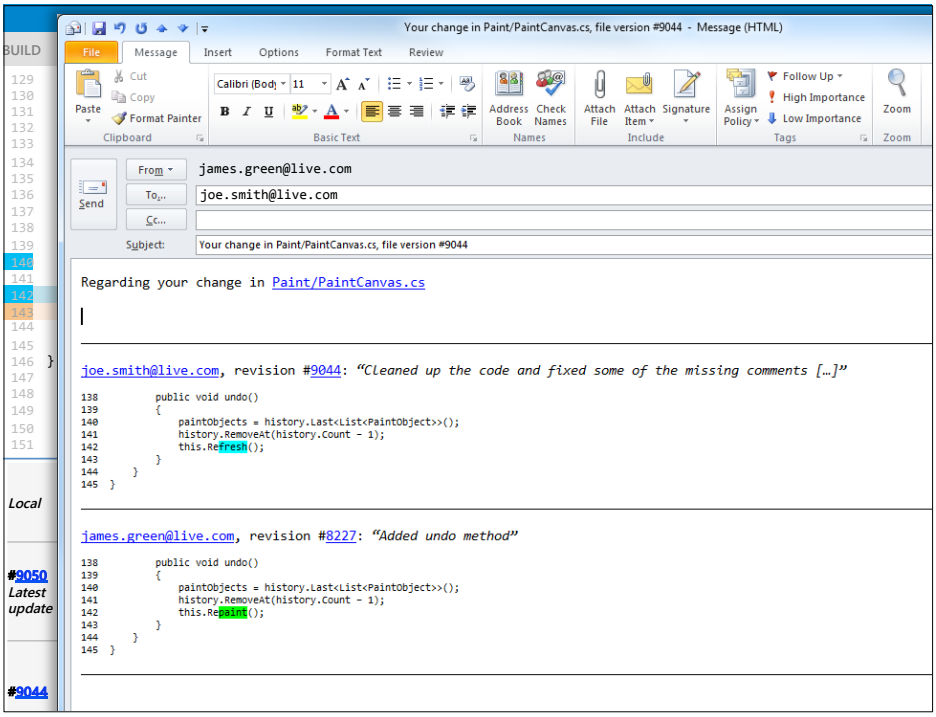


Figure 5.8: *Contacting the author of a change from within the IDE*

tion or coordinating action. In Bellevue, the communication icons (*i.e.,* email, instant message, and phone) can be used to directly contact the author of a given commit, for example to inform them about an error or a breaking change on the same code. Figure 5.8 shows the email template automatically generated by clicking on the email icon next to the author for commit #9044 (*i.e.,* Joe Smith); it includes the diff with the previous commit, for easier reference.

RITE participants' feedback—The social interaction within the view was extremely well received by all the participants. They especially appreciated the possibility of quickly using email and instant message: "*I really like that. I'd click on chat*" [R6]. When discussing what they would write to the author of the buggy change, they all specified a number things they would like to ideally see in the email, and when they saw the email automatically generated by Bellevue, they liked how it included everything they wanted: "*That is perfect.* [It is] *exactly what I would have sent*" [R1]. However, some would have liked the diff view to be as shown by Bellevue in the IDE, while "*now it's like standard diff*" [R6].

Participants' suggestions that we did not integrate during the evaluation, for time reasons, are: Adding an *email all* feature, change the title of the email giving information about

method and class in which the new change is taking place, support for copy and paste from history to email, and include more communication clients (*e.g.*, IRC or Skype).

5.4.6 Evaluation Debriefing

After each RITE session, participants were asked to fill two short questionnaires about their experience with the tool: A System Usability Scale (SUS) questionnaire [21] and a proprietary 7-point Likert scale questionnaire standardly used at Microsoft.

The SUS answers were overall positive: The mean SUS score is 85.1 (answers had $\sigma = 0.66$, on average), which is considered a high value across different domains [8, 9, 153]; as an example, the statement "*I think that I would like to use this product frequently*" scored 4.7/5.0 ($\sigma = 0.50$).

The proprietary survey was equally positive: Mean score was 5.4/7.0 (the higher the better: items only included positive wordings [154]), with $\sigma = 1$ on average. For example participants gave 5.4/7.0 ($\sigma = 1.13$) to the statement "*This product has powerful functionality and excels at what it was designed for*" and "*This product is something I am likely to share information about*" scored 5.9/7.0 ($\sigma = 0.78$).

5.5 Related Work

Coordination in software development has been studied in the fields of Software Engineering and Computer Supported Cooperative Work since the 1980s, and researchers have produced a wide range of analyses and tools [149].

Bellevue uses historical change information to support developers' coordination. Sarma *et al.* present a comprehensive review of coordination tools and defines a framework that classifies those technologies according to multiple coordination paradigms [152]. In this framework, tools such as versioning systems and issue tracking systems support the development process and are at the basis of the more sophisticated tools that provide meaningful and automatically aggregated information: These are research prototypes and industrial applications conceived to better support developers coordination in the IDE. Such tools includes full-fledged platforms, specific workspace awareness solutions, and code information visualization tools.

Full-fledged platforms, such as Jazz [92] and Mylyn [54], are at the far end of the spectrum in terms of complexity [152], and aim at transforming the IDE experience. Jazz, or Rational Team Concert, is an IDE platform, built on top of Eclipse and Visual Studio, that integrates several aspects of the software development process, including integrated planning, tracking of developer effort, project dashboards, reports, and process support. Relations between artifacts can be defined and leveraged to gather project information. Jazz also offer support for communication within the IDE (*e.g.*, instant messaging), more advanced than Bellevue. Mylyn and its successor, Tasktop Dev [167], are based on

Eclipse and Visual Studio and use task context to improve the productivity of developers and teams [101]; for example, they reduce information overload by providing developers with just the artifacts and information necessary for their current code modification task, and offer a comprehensive task repository to support teamwork by sharing information on tasks and their context. Both platforms supports the creation of novel information (*e.g.,* tasks and work items, and relations among artifacts) to support developers productivity, and encourage a task or work item based approach to evolution. Bellevue aims at using already available data and visualizing it in a non-obtrusive way. Another example of improved communication in the IDE is REmail [6], which integrates developers' email communication in the IDE to support program comprehension; REmail can be used in conjunction with Bellevue to extend the communication feature of the latter.

Workspace awareness solutions, such as Lighthouse [38], CollabVS [48], and Syde [81], are concerned with changes before they are committed to the source code repository, to address the conflict detection or real-time development information. For example, Syde tracks fine-grained real-time changes and informs developers when potential conflicts are emerging by alerting them on a view and on the code editor. Since the goal of these tools is different from that of Bellevue, they do not show change history related information.

Interestingly, Bellevue design could be included in environments such as Mylyn and Jazz, and could be used concurrently with workspace awareness tools, in order to offer coordination support from a complementary perspective.

Information discovery approaches, such as Ariadne [41] and Tesseract [150], seek and assemble information to perform tasks such as expert finding and socio-technical network analysis. Recommender systems, such as Seahawk [138, 139], exploit change information and externally generated data to support software development and comprehension. Similarly to Bellevue some of these approaches also use historical code information to inform their users. Given their goal, they offer different, complementary views on data and integration with the development environment.

Code information visualization tools include the "blame" functionality offered, for example, by git or svn.[3] This feature allows to see who did the last change on each line of code of a file, and when. Another tool is the concept presented by Rastkar and Murphy, in which the developer is able to see for a summary of commit messages connected to a line of code in the IDE [143]. In contrast, Bellevue offers an interactive view that shows detailed historical information for specific chucks with respect to a previous version. Bellevue always displays which files and lines changed, so it does not require the developer to actively ask for the commit message of the line, because the developer may not be already aware of the relevance of the file and the line. In our exploratory investigation narrowing down a breaking change to the file and line causing the issue emerged as one of the most problematic and time-consuming efforts for developers.

3. `http://git-scm.com/docs/git-blame`

5.6 Concluding Remarks

We designed Bellevue to enable developers better coordinate, by making historical information visible and more accessible in the IDE.

This chapter makes the following main contributions:

1. Requirements for a tool to support teamwork based on currently unmet usability heuristics and the results of our qualitative analysis. For example, to favor recognition of code changes over recall, and to increase the visibility of the codebase status with respect to received changes.
2. The design and evaluation of Bellevue, an IDE extension to support teamwork by improving the integration of code changes in the IDE. Bellevue makes received changes visible inside the editor, and makes the history of code chunks easily accessible using progressive disclosure.

Part III

Generating Information

6 Pollicino: Code Bookmarks

The program comprehension research community has been developing useful tools and techniques to support developers in the time-consuming activity of understanding software artifacts. However, the majority of the tools do not bring collective benefit to the team: After gaining the necessary understanding of an artifact (e.g., using a technique based on visualization, feature localization, architecture reconstruction, etc.), developers seldom document what they have learned, thus not sharing their knowledge. We argue that code bookmarking can be effectively used to document a developer's findings, to retrieve this valuable knowledge later on, and to share the findings with other team members.

We present a tool, called Pollicino*, for collective code bookmarking. To gather requirements for our bookmarking tool, we conducted an online survey and interviewed professional software engineers about their current usage and needs of code bookmarks. We describe our approach and the tool we implemented. To assess the tool's effectiveness, adequacy, and usability, we present an exploratory pre-experimental user study we have performed with 11 participants.*[1]

1. This chapter contains the paper "Collective Code Bookmarks for Program Comprehension" [77], published in the proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC 2011). The authors of this publication are Guzzi, Hattori, Pinzger, Lanza, and van Deursen.

6.1 Overview

Software engineers are often faced with the challenge of understanding a program written by someone else and a long time ago. Due to the lack of proper documentation, program comprehension may take over 60% of the software maintenance effort [33]. Program comprehension methods and techniques can have a significant impact on the overall efficiency of software developers. However, program comprehension is an individualistic and ephemeral activity. Developers create their mental models of the system [173] to perform a task, but seldom document them, *i.e.,* most of the knowledge gained during the comprehension activity resides only in the developer's mind and typically is forgotten once the maintenance task is completed. Several barriers prevent developers from documenting their findings, such as constant interruptions, outdated documents that are hard to update, and preference for face-to-face talks [110].

"Traditional" code bookmarking is a common practice among users of modern IDEs, such as Visual Studio, Eclipse and IDEA.[2] Examples of bookmarks include task markers (`//TODO` and `//FIXME`), tag annotations (@Task), and cross-reference hyperlinks [166]. They have been used to remind developers of unfinished work [166], which serves a short-term purpose; and to guide programmers to perform a task [32, 40, 128]. Most of these bookmarks are embedded in the code, which introduces a trade-off between the benefits of sharing and the drawbacks of having "noise" in the code.

We propose an approach for micro-documentation and knowledge sharing: *collective code bookmarks*. It encourages developers to bookmark artifacts while investigating the source code, and to document their findings with a description associated to a bookmark, which can be shared with other team members or maintained on the developer's workspace for private use. A collective code bookmark is a link from a specific location in a file to one or more descriptions, which may be comments, links to resources, documents, websites, and tags. Extra information can be associated with a bookmark, such as author, creation date, *etc.* Our goal is to support the benefits of collective code bookmarks, and keeping them as an external documentation.

We conducted a survey on the current usage of code bookmarks, as well as interviews with professional software engineers, to gather the requirements for collective code bookmarks and for a supporting tool to encourage developers to document their findings while performing a program comprehension task. Based on this survey, we present an approach for collective code bookmarks with the goal of helping developers to retain and share the knowledge acquired during program comprehension activities.

The contributions of this chapter are:

- Requirements for a non-intrusive bookmarking tool that facilitates knowledge sharing. These requirements are the outcome of an online survey on current usage of code bookmarks, and interviews conducted with professional software engineers.

2. See `microsoft.com/visualstudio`, `eclipse.org/`, `jetbrains.com/idea/`

- An approach to code bookmarking. We devised our approach as a publicly available tool named Pollicino. We present our approach and describe the tool.
- An exploratory pre-experimental user study with 11 participants to investigate the potential of collective code bookmarks and to evaluate the tool's effectiveness, adequacy, and usability; with promising preliminary results.

Structure of the chapter. Section 6.2 presents the related work on code bookmarks. We motivate our work in Section 6.3, where we report on requirements for a collective bookmarking tool, elicited from a survey and the interviews with practitioners. In Section 6.4 we describe our approach and the tool implementation. Section 6.5 presents the design of our experiment, while Section 6.6 reports on its results. Section 6.7 discusses the threats to validity. Finally, we summarize our conclusions in Section 6.8.

6.2 Related Work

Code bookmarking can be classified as one form of support for user-defined annotations. Other forms are task markers, tag annotations, and cross-reference hyperlinks [166]. These annotations are relatively common practice among users of modern IDEs, however their design and purpose vary greatly.

Eclipse's user-defined annotations are classified into three categories: (1) tasks—`//TODO`, `//FIXME`, `//XXX`; (2) problems—reporting invalid states of the system; and (3) bookmarks —for marking locations and having quick access to them. Tasks reside in the source code and have the purpose of reminding developers of information regarding a specific piece of code, while problems and bookmarks are external links to a specific line in a source file.

Brühlmann *et al.* proposed a generic annotation tool, called Metanool to capture and retain human knowledge during reverse engineering processes [23]. Metanool supports the association of any type of annotation (*e.g.,* comment, document, UML diagram) to a source code entity. However, it was targeted at facilitating reverse engineering activities instead of supporting active development.

TagSEA combines the notion of marking locations in spatial navigation with social tagging to support reminding and refinding (revisiting a specific part of the code) [166]. The TagSEA annotation has the form of a customized Java annotation, residing in the source code, and thus being shareable across the team. These annotations were designed to be easy for development teams to search, filter, and manage. They also proved to be useful for creating tours for technical presentations that involve interacting with the IDE [32]. TagSEA was evaluated in several (longitudinal) case studies: the findings focus on the sort of tags used and the extent to which tags could be reused.

JTourBus is a similar approach that creates tour guides to help programmers to perform a task or to assist them to get familiar with a sequence of code dependencies [128].

Contrary to TagSEA, our collective bookmarks are designed to reside outside the source code, bringing two benefits: flexible privacy setting – the author of the bookmark can decide whether to share it and with whom; and cleaner code. Furthermore, our evaluation is different, emphasizing code locations over tags, and being carried out with a larger group of participants all conducting the same tasks. Differently from JTourBus, our approach does not provide means to create guides, but a lightweight approach developers to micro-document their findings when performing program comprehension activities.

6.3 Motivation

Previous studies have reported a low use of code bookmarks that do not reside in the source code. In their report on usage data collected from developers using Eclipse, Murphy *et al.* state that only 5 out of 41 developers used the bookmark view [123]. In another survey, Storey *et al.* report that 84% of the respondents never or rarely use bookmarks [165]. Some hypotheses are raised to explain the low adoption, such as *poor visibility* in the IDE, or *poor synchronization* with the code; however no further investigation was performed to understand the reasons for the low adoption.

We conducted an online survey to investigate *why* bookmarks are rarely or never used in modern IDEs. We collected a total of 209 responses from which 71% of the participants were practitioners and 29% academics. The vast majority consisted of experienced developers (60% had more than 6 years of experience, 30% had 3-5 years of experience, the rest had less than 3 years of experience).

From the respondents, 88% report that they never or rarely use code bookmarks (confirming previous results). We furthermore asked them why this is the case. Among those 88%, who never or rarely use bookmarks, 50% answered they did not know that bookmarks existed in their IDE, 25% stated that they do not find them useful, while 9% think creating a bookmark is cumbersome (Figure 6.1).

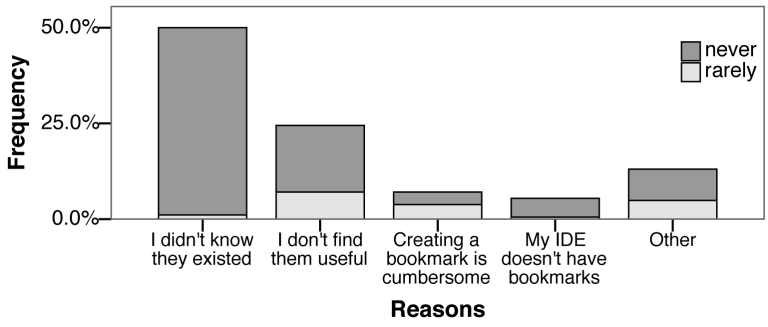


Figure 6.1: *Reasons for not using bookmarks in the IDEs*

We also collected qualitative information about how people are using bookmarks. A number of respondents reported that bookmarks are useful for keeping track of points of interest in the code while understanding it ("*I use them while I'm skimming code and trying to get a clue about how to fix something. I therefore bookmark interesting pieces of code to be further inspected*") and to quickly access them later ("*as a way to return back to the same bit of code a few days later*"). Respondents also reported to use them as a guide while working on a task ("*For marking something of interest, so I don't lose it. Shift-F11 in IDEA lists all bookmarks. I tend to create a series of bookmarks for a particular task, so I can find the main bits of the problem.*").

To further investigate the potential of our approach, we conducted interviews with four practitioners with different roles: (R1) a software developer with 6 years of experience who was not aware of code bookmarks; (R2) a software developer with 8 years of experience who was aware of code bookmarks; (R3) an open-source developer 10 years of experience; and (R4) a software architect and team leader with 25 years of experience.

R1 reported that his team has the culture of placing regular comments and task markers in the code to remind, refind, and annotate decisions. This is a problem when the code goes to quality assessment: "*In the system we work we see thousands of TODOs, and we never do anything about it.* […] *Every six months the manager freaks out because of the quality control, and we have to clean up the entire code.*"

R2 actively uses code bookmarks, mainly to understand a legacy system that his team maintains. He reported that system size and lack of documentation are the main reasons for using bookmarks: "*The code is huge and I really need them as breadcrumbs, in particular when I'm digging into code.* […] *I'm changing code that was written by 5-6 people in 10 years with no architecture, no design,* […] *and I must have a way to track all the jumps that I'm doing.*" One of his main complaints was the lack of share-ability of bookmarks: "[…] *and I actually tell him (teammate) where to add his bookmarks* […] *but since up to now there's no way of sharing them, most information remains just for me.*" Since R2 is actively using code bookmarks, we asked him what features he would like to have, and his answer was: "*I want to share my bookmark's description, and I'd like to add resources to it.* […] *Keep it simple.*" Other suggestions were to offer grouping options, and sequencing by the user's choice.

With R3 and R4 we investigated when collective bookmarks would be useful. R3 thought they could be very useful to assist the developers of his team, who are spread across different locations. R4 identified the following situations: when a developer is passing the responsibility of a part of the code to another person; to maintain traceability between UML diagrams and the code; in the beginning of development, to annotate the most important methods of the API.

Summary

Based on the survey and the interviews, we have identified the following *requirements* for collective code bookmarks and for a tool that supports them.

Be share-able. Current bookmarks are either private or can be shared through the source code, imposing a restriction on how a developer can share it. The feedback collected suggests that developers want to be able to share their bookmarks, but doing it so through the code clutters the source code.

Have a description. Code bookmarks should be used to micro-document findings. Having a textual description of each finding is essential for reminding it later.

Support grouping and sequencing. Developers would like to be able to associate auxiliary information to bookmarks to help organizing them.

Be platform independent. There are numerous languages and IDEs, and having a bookmark model for each combination restricts the social benefit of bookmarks. There should be a standardized model that can be used in any language/IDE.

Be simple to use. The action of bookmarking a location in the code should be non-intrusive, intuitive and almost effortless. Otherwise the gain from the information registered is hindered by the effort to register it.

6.4 The Pollicino Approach

6.4.1 Collective Code Bookmarks

Two requirements for collective code bookmarks are to be shareable and platform independent. Thus, we redefine code bookmarks by proposing a meta-model, in the form of a standardized and extensible XML schema. Using an XML schema to encode the above information allows code bookmarks to be shared as XML files, making them portable to any IDE and attachable to any source file.

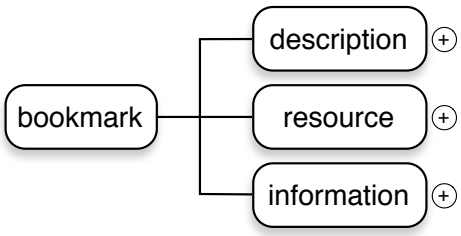


Figure 6.2: *Meta-model for collective code bookmarks*

Figure 6.2 illustrates our meta-model, composed of three parts. The *description* can be a text message, a link to internal/external resources, or tags. Any type of description can be added (ranking attributes, pictures, movies, etc.). The *resource* contains information about the location of the bookmark in a file: project—the name of the project; location—the

full path of the file within a project; line number—the line number where the bookmarks should be placed. The three attributes are mandatory to properly locate the bookmark in a file. Finally, *information* contains an extensible list of optional attributes (such as author, creation time, group, or a file revision number).

6.4.2 Pollicino

To assess the feasibility and usefulness of collective code bookmarks, we have implemented Pollicino[3] (see Figure 6.3), an Eclipse plug-in that allows developers to place bookmarks as a way to micro-document their findings, and share them with others. In the following, we present the main features of Pollicino.

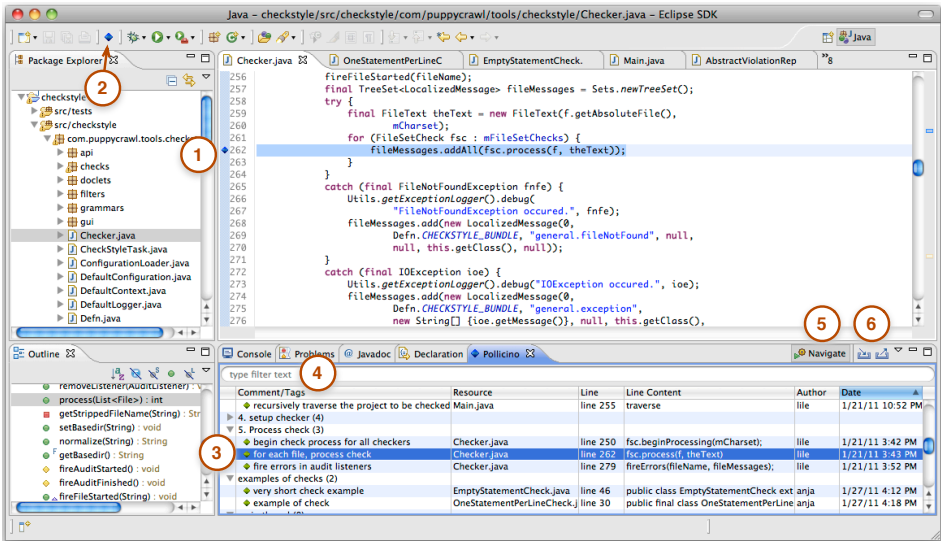


Figure 6.3: *Eclipse with the Pollicino plug-in*

Bookmarking the code – To bookmark a code snippet, the user can mark it and right-click on the editor or on the left ruler, and choose "Add Stone" in the popup menu. The user then enters the minimum information (*e.g.*, a description) after which Pollicino inserts a blue marker into the left ruler at the corresponding line (Point 1). Bookmarks can be added in any type of file (*e.g.*, source code, text file, XML file, and build file). Some participants of the survey reported that creating bookmarks is cumbersome. To alleviate this, the user can create bookmarks with a simple keyboard shortcut, easing the mechanical process of creating bookmarks.

3. The name *Pollicino* ('Little Thumbling' in Italian) is inspired by the fairy tale Hop-o'-My-Thumb, in which a boy uses pebbles to find his way back home.

Showing bookmark information – The Pollicino view can be displayed with one click (Point 2), showing the bookmarks that are currently in the user's workspace and the information associated with them (Point 3). The view shows the user's comment, the resource and the line where the bookmark is located, the line content, the author and the date.

Grouping and sorting bookmarks – There are different ways to organize bookmarks on the view. They can be grouped by "active/archived", or the user can create customized groups. The user can also sort bookmarks according to any column of the view. The information in all the columns can be searched for specific information using the search bar (Point 4).

Navigating through bookmarks – Double-clicking on a bookmark entry in the view takes the user to its location in the editor. The view also provides a *Navigation* mode (Point 5), which shows the location of a bookmark with a single click (or navigating with up and down arrows).

Sharing bookmarks – It is possible to import/export all or a selection of the bookmarks from/to an XML file that follows the schema presented in Section 6.4 (Point 6).

Customizing bookmarking – When creating a bookmark, by default the user has to enter a comment and her name. The user can transform the creation of bookmarks as a one-step process by customizing the information she needs to enter on the preferences page (*e.g.,* save her name and add a blank comment).

Pollicino is available at `http://www.st.ewi.tudelft.nl/~guzzi/pollicino/`.

6.5 Pre-experimental study design

We conducted a user study with the goal of investigating the potential of our approach. To achieve our goal, we assess the tool's *effectiveness, adequacy and usability* using an exploratory pretest-posttest pre-experimental design [25]. Pre-experimental indicates that the experiment does not meet the scientific standards of experimental design [3], yet it allows to report on facts of real user-behavior, even those observed in under-controlled, limited-sample experiences.

The study is composed of several experimental runs. Each run consists of the initial pretest questionnaire, a demo/tutorial of Pollicino followed by three assignment tasks (to perform within Eclipse with our plug-in installed), a final posttest questionnaire and a concluding debriefing talk. The complete handout used in the experiment, including the tasks, the pretest and posttest questionnaires, can be found in our on-line appendix [78].

6.5.1 Research Questions

The main goal of the experiment is to investigate whether Pollicino can help software developers during program comprehension activities (*effectiveness*). Our main research question is:

[RQ1] Can collective code bookmarks help developers to perform program comprehension activities?

We identify the following three sub-research question:

[RQ1.1] Can Pollicino be used to (micro-)document a developer's own findings?

[RQ1.2] Can micro-documentation via bookmarks be useful to team members to get starting points (*i.e.,* entry points for program comprehension)?

[RQ1.3] Can Pollicino be useful during development tasks?

A secondary goal of the experiment is to gather initial feedback on our tool. In particular we are interested to explore whether the tool's outcome matches the user's expectations (*adequacy*) and whether the tool is easy to use (*usability*). We thus identify the two additional research questions: (**RQ2**) Does Pollicino match the expectation for a code bookmarking tool? and (**RQ3**) Is Pollicino sufficiently usable by developers? Finally, we also want to gather feedback on how to improve Pollicino.

6.5.2 Pretest-Posttest Design

In a pretest-posttest study, participants are subject to a first test (questionnaire), before performing the experiment, and to a second test, after performing the experiment. For most questions in our questionnaires, we make use of closed-ended matrix questions: participants are asked to rate a number of statements on a five-point Likert scale.[4]

Pretest Design

The pretest questionnaire is split into 5 parts. The first two parts are dedicated to understand the personal background (part 1) and the software development experience (part 2) of our participants.

In part 3 we ask participants to rate a number of statements regarding their habits when understanding code. A participant's habits might influence her expectations with respect to a tool that eases program comprehension. We thus investigate common practices, *e.g.,* reading the comments inside the code, reading the available documentation, and making

4. 1 = "strongly disagree" , 2 = "disagree" , 3 = "neither disagree nor agree", 4 = "agree", 5 = "strongly agree".

Table 6.1: *Statements used in the pretest to measure adequacy*

a)	Such a tool would prevent me from getting lost in the code
b)	I don't see the added value of such tool
c)	Bookmarking would help me when I'm trying to understand a functionality
d)	As soon as I understand the code, bookmarks become useless to me
e)	Such a tool would help me manage points of interest in the code
f)	The tool will not be able to help me in real problems
g)	My bookmarks will help others understand what I did

changes to the code and run it to see what happens. Similarly, in part 4, we try to assess the participant's familiarity and attitude toward code bookmarking. Participants are asked to rate dedicated questions, according to whether or not they were aware of the bookmarking feature of Eclipse. Additionally, an open ended question asks them to provide alternatives that could replace bookmarks in their function of *code location markers*.

In part 5, participants are given an abstract description of Pollicino (see [78] for the full text), and asked to rate a number of statements about their expectations of a tool with such functionalities (see Table 6.1). These statements are then asked again in the posttest, after introducing the independent variable (*i.e.,* the use of the tool), to assess adequacy.

Posttest Design

The posttest questionnaire is given to the participants after they performed the assigned tasks to, *e.g.,* verify whether Pollicino is seen as a good bookmarking tool.

We split the posttest into 5 parts. In part 1, we ask the participants to assess a few statements about their general experience in performing the experiment, *e.g.,* whether they found the three tasks doable and whether they felt time pressure.

In part 2 of the posttest we ask participants about their experience with Pollicino while performing the tasks. In particular, we collect information on the tasks and purposes the tool has been used for. We use the results from this part to address **RQ1**.

Part 3 of the posttest is dedicated to assess the participants' perception of Pollicino. Participants are asked to rate similar statements as in question 5 of the pretest. The difference is that, at this time, we ask directly about the tool and not about code bookmarking in general. By comparing answers to this question and to its counterpart in the pretest, we can verify how Pollicino meets the participants' expectation and address **RQ2**.

In Part 4 participants are asked whether they used particular features of the tool (*e.g.,* key binding to add a marker) and, if yes, to indicate how useful they were on a scale from 1 = "very useless" to 5 = "very useful". The questions in part 5 are used to measure the usability of the tool. These measures are used to address **RQ3**. In addition, these statements allow

us to verify whether usability issues might have influenced the overall experience with the tool during the assignment.

After the posttest questionnaire, we held a debriefing talk to collect additional information, both on the tool and the experiment. During this individual talk with participants, we could better understand about their experience with the tool and ask them about their opinion on possible improvements.

Checkstyle

The system we chose as object of our experiment is *Checkstyle*.[5] We used version 5.3, which consists of 341 classes distributed across 22 packages, for a total of 46 KLOCs.[6] Our choice was motivated by the following factors: Checkstyle's size allows for performing an experimental session, yet being representative of real life programs. It is written in Java, with which many potential participants are sufficiently familiar. It has been used in previous experiments [172, 178, 34], from which we could reuse one of the tasks.

Tasks

We designed the tasks for our experiment according to the sub-research question related to **RQ1**, from which we derive the following three different scenarios: (1) investigate and understand a part of a system, (2) (micro-)document a system's functionality, and (3) add functionality to the system. The scenarios require a program comprehension process and are inspired by Pacione's taxonomy [134].

Task T1 is used to address **RQ1.2**. It requires the participant to gain general knowledge about the execution stages of Checkstyle, and is reused from Cornelissen's controlled experiment [34]. To simulate a collaborative environment, a number of code bookmarks were already placed in the project. T2 is used to address **RQ1.1**. It encourages the participant to understand and simultaneously document (by adding bookmarks) the class hierarchy related to the functionality that checks the adherence to each code convention. The last task T3 is focused on implementing a new check, to which the knowledge obtained and the bookmarks added in the previous two tasks is potentially useful. We use this task to address **RQ1.3**.

The three tasks were tailored to be feasible in the allotted time (20 minutes for each task) considering the minimal experience level required from the participants. For more information on each task we refer the reader to [78].

Pilot Studies

We conducted four pilot studies to test the experiment's feasibility and duration. With the first two runs we found out that two of the tasks needed to be more focused and

5. See `http://checkstyle.sourceforge.net/`
6. Measured using `http://eclipse-metrics.sourceforge.net/`

have more guidance to be doable in the allotted time. We furthermore identified some defects in Pollicino, which were fixed before the actual experiment. After the third test run, we adjusted a few statements from the questionnaires. The fourth pilot study ran without problems.

6.6 Results

We report on the results obtained from our experiment. We first describe our participants and their attitude toward program comprehension and code bookmarks (as measured from the pretest). Then, we report on their performance during the assignment (as measured from the posttest and from analyzing the code bookmarks placed during the experiment). Finally, we present the results from the posttest, along with the feedback from the participants, to answer our research questions.

6.6.1 Participant Characteristics

Twelve volunteers participated in our experiment. Two participants were from the University of Antwerp, two from the University of Lugano and seven from the Delft University of Technology. Four participants were PhD students, seven participants were MSc students (three of which are working as part-time developer). All participants were male, aged between 24 and 30. One participant reported that the tasks were not feasible. Since task feasibility was a requirement for our experiment, we excluded his results from our analysis, giving a total of 11 subjects in our study.

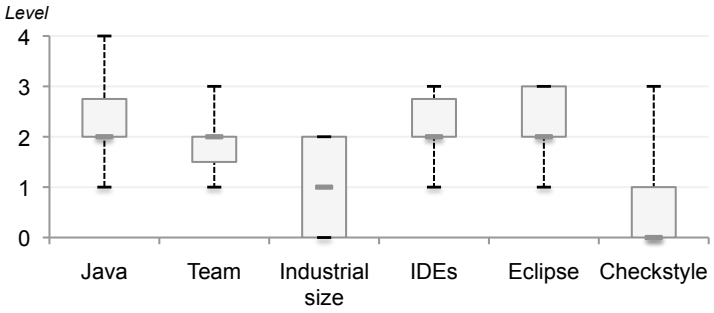


Figure 6.4: *Participants' experience*

Figure 6.4 reports the participants' experience regarding software development on a 5-point scale.[7] The box plot depicts the following data: minimum value, lower quartile, me-

7. 0 = "does not know the subject", 1 = "familiar with the subject, but still have some difficulties with it", 2 = "comfortable in the subject and currently using it daily", 3 = "highly proficient in this subject", 4 = "serves as reference for colleagues, and feels confident in helping them".

dian, upper quartile, and the maximum value. As can be observed, participants reported to have good knowledge of Java, IDEs, and Eclipse, while the experience in working on industry-sized systems is lower. All participants reported to have no or low experience with Checkstyle, indicating that their answers were were not influenced by previous knowledge about Checkstyle.

6.6.2 Comprehension Attitude

In the pretest, we ask the participants to rate a number of statements aimed at measuring their attitude toward both program comprehension and bookmarking. We next go into detail of these two aspects.

Attitude toward program comprehension

The most popular practices (indicated by 10 out of 11 participants) when working on a task on unfamiliar code of a system are to investigating the code before starting with the task, and to reading the code comments (if available). Most of the participants recognized that they often navigate through several classes and find themselves lost amongst many open tabs. Seven participants stated that they read the available documentation, and make changes and/or add print statements to the code and then run it to see what happens. Some less popular practices indicated are: to start looking at available test suites, to use specific tool or markers (*e.g.,* `//TODO`), or to look at UML diagrams.

Overall, we can say that our participants' effort in program comprehension relies mostly on investigating the code, reading comments and documentation (when applicable), and making changes (or insert print statements) before running the code.

Attitude toward (code) bookmarking

One participant who knew about code bookmarking within Eclipse indicated that the feature is relatively easy to use. According to him, creating a bookmark is not particularly cumbersome and code bookmarks are useful. Another participant did not think that bookmarks are easy to use. He indicated that creating code bookmarks is cumbersome and he is not sure whether they are useful.

Seven out of the nine participants who did not know about this feature, indicated that they agree with the statement "Code bookmarks seem to be useful". The remaining two indicated they "neither disagree nor agree" with the statement.

We asked participants to shortly report on which tools or methods they use to function as "code location markers" during their program comprehension routines. One participant reported that he makes use of visually outstanding comments (*e.g.,* "`//************//`"), so that the location is easily spotted when looking at the code. Alternatively, he introduces lots of consecutive new lines, to create a easily noticeable blank space. Another participant

Table 6.2: *Examples of bookmarks placed during the experiment*

Comment	File	Location
Use this method to specify which tokens to respond to	Check.java	getDefaultTokens()
Implement which token the check is interested in	Check.java	getDefaultTokens()
If you're interested, this is where your Check is actually called	TreeWalker.java	notifyVisit(DetailAST ast)
Data structure, it's parsing a tree	DetailAST.java	class declaration
This class must be extended for Format checking purposes	AFormatCheck.java	class declaration
Use this class for logging check output	AViolationReporter.java	class declaration
Checks respond on tokens. These are the types you can respond to. Compare this to the Listener system: you subscribe to all tokens you want to respond on	TokenTypes.java	class declaration

explicitly introduces compilation errors and also uses a custom tool. One participant wrote that he uses Mylyn to link code to tasks.

Overall, most participants were not aware of code bookmarking features, yet acknowledged the usefulness of code location marking facilities.

6.6.3 Task Performance

During the assignment, the participants were invited (but not required) to use the bookmarks already placed in the project, and to add their own bookmarks. Table 6.2 illustrates some typical bookmarks they added while performing the tasks. Observe that several bookmarks were placed at the declaration of a class with a short description of its purpose. Other locations where bookmarks were commonly placed are at the method declaration, with a short description of its functionality; or at some statement inside a method, where there is a method call of interest.

For T1, we provided a set of bookmarks that was split into 5 groups, which corresponded to the main stages of a check in Checkstyle. Most participants seem to have benefitted from the grouping, since 10 out of 11 correctly identified the stages, which varied from 4 to 6 in their answers.

T2 required understanding the Check hierarchy, placing bookmarks during this process. The number of bookmarks placed by the participants is shown in Table 6.3. A common practice was to look for the abstract implementation of a `Check` that could be extended, or an example of its implementation, and mark its location.

For T3, we asked the participants to implement a check to count the number of methods in a class. Two participants had correct implementation, three copied the content of a similar class (making the implementation also correct), two had incomplete implementation on the right track, and four had incorrect implementation. We did not identify a correlation between the number and quality of bookmarks added in T2 and the quality of the code implemented for T3.

Table 6.3: *Number of participants who placed a certain number of bookmarks for T2*

# Participants	2	2	1	1	3	2
# Bookmarks	8	6	5	4	3	0

Overall, participants reported that the tasks were feasible, interesting, and realistic, that the warm up task (a short "hands-on", which preceded the tasks) was useful, and that the experiment was fun to do. Four participants indicated that they would have needed more guidance to complete the tasks. Eight participants felt time pressure.

6.6.4 Experience with Collective Code Bookmarks

In this section, we discuss and answer **RQ1**, which is associated with the effectiveness of Pollicino in helping developers to perform program comprehension activities. We first address the three sub-research questions related to it.

RQ1.1: Can Pollicino be used to (micro-)document a developer's own findings?

Participants reported that Pollicino was useful during T2 (six participants reported to have used the bookmarks already contained in the code), where the scenario was to (micro-)document some of the system's functionality. In addition, participant feedback on Pollicino's bookmarks was generally positive. They found the navigation of bookmarks useful, as they could "*tag important stuff and then* [...] *quickly navigate to it later*" (P6). P8 hypothesizes bookmarks could be used as a mind map for a developer to document his findings. P2 sees "*a very natural relationship between* Pollicino*, code, and diagrams*", and suggested the possibility to link Pollicino's bookmarks with design documents (*e.g.,* UML diagrams). P7, who has work experience, said his practice is to annotate code via comments and then search for them. After trying our tool, he thinks "*a bookmark would be very handy for that*".

We conclude that Pollicino can be used to (micro)-document a developer's own findings.

RQ1.2: Can micro-documentation via bookmarks be useful to team members to get starting points?

Participants reported that Pollicino was useful during the first task T1, where the scenario was to investigate and understand a part of a system, and ten of them had correct answers to the task. Additionally, during the debriefing talks, participants recognized the value of using Pollicino within a team, *e.g.,* by having step-by-step instruction to help newcomers steer their way in a project (P9). They also stressed the importance of sharing and synchronization of bookmarks, recognized the need of supporting synchronization of bookmarks, proposing to have bookmarks automatically integrated with a version con-

trol system. Not all participants are convinced that existing bookmarks helped them to understand the system. They emphasized that a bookmarks' description should be as meaningful to other people as to its author.

We conclude that Pollicino can be useful for team members to get starting points.

RQ1.3: Can Pollicino be useful during development tasks?

Only 2 participants reported Pollicino as useful during T3, which asked to implement a functionality in the system. Also, there seem to be no correlation between the number of bookmarks placed and the quality of the code implemented for T3. A few participants mentioned during the debriefing talks that Pollicino bookmarks could be used during development tasks, *e.g.,* as a replacement for "temporary" `TODO`s (P3). Participants were not convinced about the potential of Pollicino during development.

We conclude that regarding active development, Pollicino is not as useful as for documenting or understanding code.

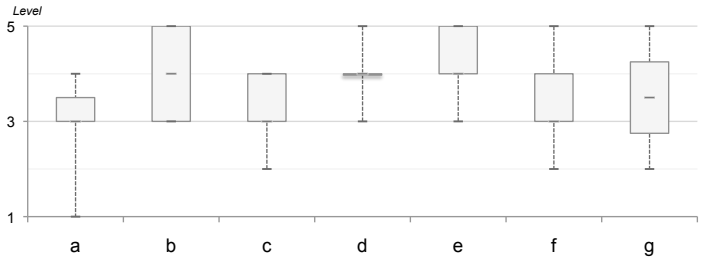
Summary

Overall, participants found our tool useful when their tasks were precisely to understand or document the code (nine and eight matches, respectively), while they did not find it useful during implementation (only two found it useful). This observation reflects on the use of bookmarks, which were mostly added during the micro-documenting task, as indicated by ten participants. Therefore, we can answer **RQ1** positively by stating that collective code bookmarks can help a developer during the documentation of her own findings, and this information can be useful to her team members.

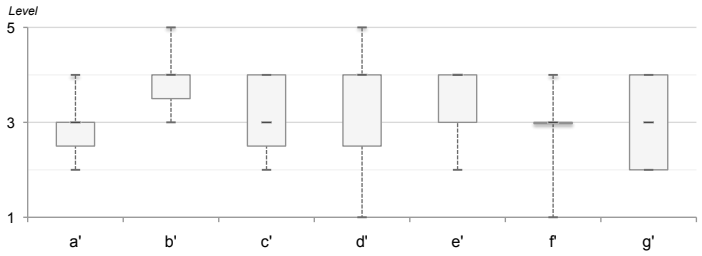
6.6.5 Expectations vs. Perception of Pollicino

To answer **RQ2**, associated with the *adequacy* of our tool, we analyze of expectations *vs.* perception of Pollicino. In the pretest, the participants rated seven statements about a hypothetical tool with Pollicino's functionality (see Table 6.1). In the posttest they rated the same statements, but this time about Pollicino itself. By comparing the answers to this question and to its counterpart in the pretest, we can verify how the perception of our tool meets the participants' expectation.

Our sample is composed of (the ratings of) 11 participants. The box plots in Figure 6.5 depict the following data from our sample: minimum value, lower quartile, median, upper quartile, and the maximum value. To ease the visualization and analysis of the data, we mirrored the ratings of statements *b*, *d* and *f* (*cf.* Table 6.1) that were formulated with a negation (*e.g.,* "4 = agree" becomes "2 = disagree"). We did the same for the ratings of the corresponding statements in the posttest.



(a) *Expectations, as measured in the pretest*



(b) *Perception, as measured in the posttest*

Figure 6.5: *Expectations and perception of a tool like Pollicino*

Figure 6.5a summarizes the ratings obtained for the expectations of a bookmarking tool. Thus, in advance many participants are not sure about whether the tool can prevent them from getting lost in the code (*a*). Most participants are positive about the added value of the tool (*b*), its value to others (*d*), and about helping someone to manage points of interest in the code (*e*). Participants were less, but still positive about the tool helping them while trying to understand a functionality (*c*), especially in real problems (*f*), and also helping others to understand one's findings (*g*). In general, the expectations of a bookmarking tool with collective benefits were very positive.

Figure 6.5b summarizes the rating levels we obtained for the perception of Pollicino. We can see that after using Pollicino, the perception is positive for most participants and in most aspects. This is reflected by the fact that most participants see the added value of Pollicino (*b′*), think that the tool did not become useless when they understood the code (*d′*), and believe that the tool helped them to manage points of interests in the code (*e′*). Participants are still not sure whether Pollicino was able to help them with real problems (*f′*), and prevented them from getting lost in the code (*a′*). They have different opinions on whether their bookmarks will help others understand what they did (*g′*).

Overall, the perception of the participants remained positive, although the comparison of the box plots in Figure 6.5 suggests that Pollicino does not match the high expectations of the participants. To test whether this difference is significant, we performed a Wilcoxon signed-rank test with the ratings from the paired statements. Wilcoxon is

a non-parametric test to assess the null hypothesis that the medians of two samples do not differ. The results show that for each pair the Wilcoxon test is non-significant (p-value>0.05), hence from this data we can not conclude a difference between the expectations and perception of Pollicino.

Therefore, we answer **RQ2** by arguing that, even though the results of the comparison suggest that Pollicino did not match the expectations of a collective code bookmarking tool, the Wilcoxon test did not show a statistical significance on the difference between expectations and perception. During the debriefing talks, some of the participants argued that they were expecting the tool to teach them how Checkstyle works (*e.g.,* P8 said that "*they are a starting point, but they don't teach you the system*"). Instead, the goal of Pollicino is to guide its users to points of interest while understanding a concept of the system. Hence, there was some mismatch between what the tool could offer and what the participants were expecting from it.

6.6.6 Tool Feedback

Pollicino Usability

Of the eleven participants, nine found Pollicino easy to use, while the other two rated the statement "*I found Pollicino's Eclipse extension easy to use*" with "neither disagree nor agree". Participants are generally positive that they would be able to get used to Pollicino during their everyday work activities. Participants reported they did not get error messages while using the tool and that defects they possibly experienced did not severely hinder its usefulness. Furthermore, no participant reported that the information from the tool distracted him from the tasks.

The feedback on the tool's usability was positive: participants liked the simple, but complete and easy to use, interface of Pollicino. P2 said that "*everything I needed was there*", while P5 observed: "*It's a basic Eclipse feature, so it's easy to use and understandable*".

Based on this, we can answer **RQ3**, regarding Pollicino's *usability*, positively.

Pollicino features

We also wanted to gather feedback on how to improve Pollicino. We focus here on the tool's features. Their perceived usefulness was measured in part 4 of the posttest, while additional feedback and suggestions were collected during the debriefing talks.

All eleven participants indicated that the possibility of grouping bookmarks and the Pollicino view are "useful" and "very useful". The same rating level was given to both means to add a bookmark in the IDE (*i.e.,* the popup menu and the keybinding), for which most participants only used the popup menu, two participants used only the keybinding and one reported to have used both (finding both of them useful). Moreover, participants find the navigation mode in the view, used by nine of them, and the sharing (import/export)

of bookmarks also useful. Eight participants used the option to archive and activate bookmarks, and they have different opinions about whether it is useful or not, with a tendency to find the feature not so useful.

In the debriefing talks, participants suggested ways to improve Pollicino. In summary: have a custom order for the bookmarks within groups; be able to hide archived bookmarks; have the possibility to add a new empty group; have the possibility to create a hierarchy for bookmarks (per user, per package, per class, *etc.*); be able to add multiple bookmarks to one location in the code; have different type of bookmarks (*e.g.,* text comment, example, and todo) to further categorize bookmarks, and maybe have different colors for each type.

6.7 Threats to Validity

Internal Validity

Participants. To ensure the minimal knowledge required to perform the assignment, we asked them to rate their expertise on a number of topics related to the experiment. In addition, feasibility of the tasks was a requirement, and the one participant who felt the assignment was unfeasible has been excluded from the analysis.

Questionnaires and Tasks. The 5-point scale questions may have influenced the participants to follow a pattern on assigning points to the statements. We interleaved affirmative and negatory statements to mitigate this effect. Since some participants could have known that the tool was created by the researchers performing the evaluation they could have been affected by the moderator acceptance bias [63]. Providing a description of a collective code bookmarking tool, we may have influenced the participants to think our tool would teach them about the object system, which may have had a negative influence on the comparison between expectation and perception of our tool. Each task was associated to one program comprehension activity for which Pollicino may be helpful. For one of these, the authors added bookmarks that might have made it too easy to answer.

Experimental runs. There were several runs, and differences between them, such as different training of Pollicino, may have influenced the results. To alleviate this, we ran 4 pilots to fine tune the experiment, and followed a defined script when giving the tutorial.

External Validity

Participants. The fact that the participants were from academia may have limited our ability to generalize the results to the industrial environment. To alleviate this effect, we made sure participants had a minimum knowledge of the related topics, and felt the tasks were feasible. Also, a number of participants have experience as practitioner.

Tasks. Our choice of tasks may not reflect real questions related to program comprehension. To mitigate this threat, our tasks were inspired by Pacione's taxonomy [134], and task T1 was reused from a previous program comprehension experiment [34].

Object System. Even though Checkstyle is a largely used open-source system, it may not be representative of complex commercial systems. Thus, the use of a different object system may have yielded different or more reliable results.

6.8 Concluding Remarks

We have reported on the results of a survey with 209 respondents and on interviews with 4 practitioners. After eliciting the requirements for a collective code bookmark approach, we presented Pollicino. We reported on a pretest-posttest pre-experimental user study to assess the effectiveness, adequacy, and usability of Pollicino. 11 subjects participated in our user study, which consisted of: performing three program comprehension tasks (using the Eclipse IDE with Pollicino installed), answering two questionnaires (one before and one after the tasks), and having an individual semi-structured debriefing interview.

The results illustrate that Pollicino can be effectively used to (micro-)document a developer's findings, and that those can be used by others in her team. However, the tool was not effectively used for program comprehension during active development. This could be partially due to the need of adjusting one's work habits, when a new tool is introduced. We also assessed and concluded that Pollicino is usable. Directions for future work are to improve Pollicino (*e.g.,* to better support synchronization of bookmarks, for example by having the code itself carry the bookmarks), to investigate better visualizations of the bookmarks (*e.g.,* inline within the code), and to perform a longitudinal study to assess its value and that of collective code bookmarks.

7 James: Micro-Blogs

Software engineers spend a considerable amount of time on program comprehension. Although vendors of Integrated Development Environments (IDEs) and analysis tools address this challenge, current support for storing and sharing program comprehension knowledge is limited. As a consequence, developers have to go through the time-consuming program understanding phase multiple times, instead of recalling the knowledge from their past or other's program comprehension activities.

In this chapter, we aim at making the knowledge gained during the program comprehension process accessible, by combining two sources of information. Inspired by the success of Twitter, we first encourage developers to micro-blog about their activities, telling their team mates (as well as themselves) what they are working on. Then, we combine these short messages with automatically collected interaction data on, e.g., classes, methods, and work products inspected or modified by developers. We present the underlying approach, as well as its client-server implementation in an Eclipse plugin called James. We conduct a first evaluation of its effectiveness, assessing the nature and usefulness of the collected messages, as well as the added benefit of combining them with interaction data.[1]

1. This chapter is an extended version of the paper "Combining Micro-Blogging and IDE Interactions to Support Developers in their Quests" [79], published in the proceedings of the 26th International Conference on Software Maintenance (ICSM 2010). The authors of this publication are Guzzi, Pinzger, and van Deursen.

7.1 Overview

To conduct a software maintenance task, software developers need to build up a substantial amount of knowledge about the software being changed [33]. For example, developers need to understand dependencies between classes, the impact of changes to particular methods, or the ways in which two services interact.

Once the maintenance task is completed, most of the knowledge built up during the process of conducting the task will "disappear": The only permanent result is the modified software, and, optionally, some updates made to the requirements or (UML) design documentation. This is an unfortunate situation, since this knowledge may be valuable for future maintenance tasks, possibly conducted by different developers. In our research, we seek ways to avoid this loss of precious knowledge.

A solution that would require developers to extensively document their findings while working on the system is likely to fail: This would substantially slow down the completion of maintenance tasks, which is usually unacceptable; moreover the knowledge gained would be relative to the current state of the code and, thus, ephemeral. Therefore, we must seek for lightweight, unobtrusive forms of information recording. In this chapter, we investigate two such forms, *i.e.,* micro-blogging and IDE interaction collection, and study their combination in particular.

In Web 2.0 applications such as Twitter and Facebook, users provide status updates to their friends and followers, informing them about what they are doing. These applications are tremendously successful, and one of the questions that we try to answer in this chapter is to what extent similar forms of micro-blogging can be used to update team members in a software development project about what is happening in the project. To that end, we propose a novel lightweight approach that integrates (Twitter-like [132]) micro-blogging into the Integrated Development Environment (IDE): We first of all encourage developers to micro-blog about their activities; furthermore, we propose to combine these short messages with interaction data automatically collected from the IDE. One could say that we add "location awareness" to the messages by recording which, *e.g.,* classes, methods, and work products are inspected or modified by the developer.

In this chapter, we present a way to collect user actions, group them into cohesive *interactions*, and combine them with messages into what we call a *quest* (Section 7.2). We describe a prototype tool we built, called James, which includes an Eclipse plugin allowing developers to write and view new messages, and which collects IDE events triggered by the developer (Section 7.3). Furthermore, using James, we conducted a set of explorative user studies (Section 7.4), in which we evaluate (1) to what extent developers are willing to communicate their activities through micro-blog messages; (2) the sort of information they typically provide in the messages; and (3) the quality of the connections between messages and interactions as established by our algorithms. The results of our explorative studies provide strong indication of the great potential in the combination of micro-blogging and automated collection of IDE interaction data.

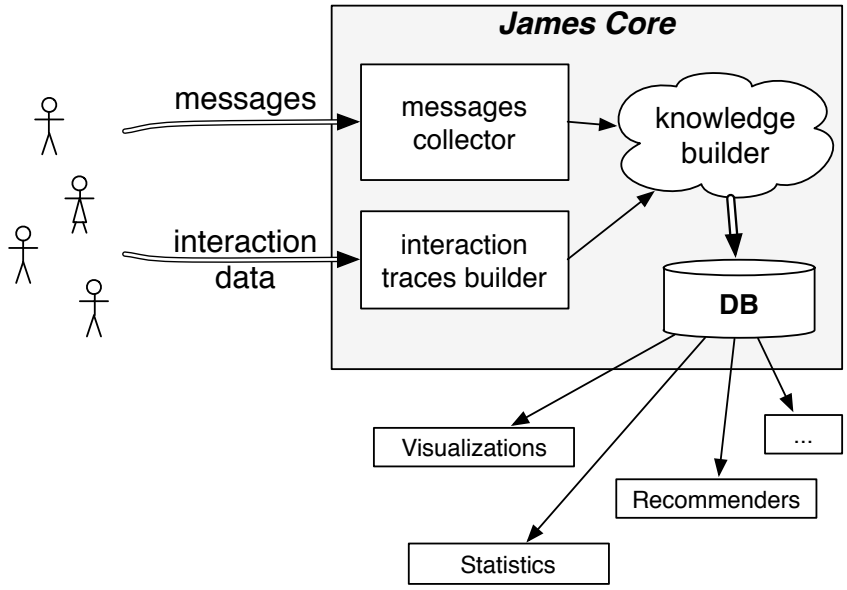


Figure 7.1: *Overview of the James approach*

Natural next steps for our work are to share the knowledge thus collected with among all the team members, to integrate other elements of social networks into the IDE, such as the ability to (un)follow team mates, specific projects, packages, or classes, and the adoption of recommender systems based on interaction and micro-blogging histories [171]. We briefly discuss these subsequent steps of our work in Section 7.5. The focus of the present chapter is on the messages themselves and their connections to IDE interactions, providing a necessary first step towards such a collaborative development environment.

7.2 Approach: Quest = Message + Interactions

We aim at combining messages and IDE interactions to record knowledge built up during software maintenance tasks. We discuss how we collect and group interaction data, and how we expect developers to report on their activities.

The overall approach is illustrated in Figure 7.1. Developers interact with their IDE as they normally do, resulting in navigation data collected as interaction traces by an IDE plugin. Furthermore, developers can actively write (short) messages, which are also collected by the IDE. Both data sources are linked to each other and stored in a repository. We refer to this combination as *quest*. The stored data can subsequently be used in visualizations, recommendations, or other presentation forms helpful to developers. The schema used in the repository is illustrated in Figure 7.2.

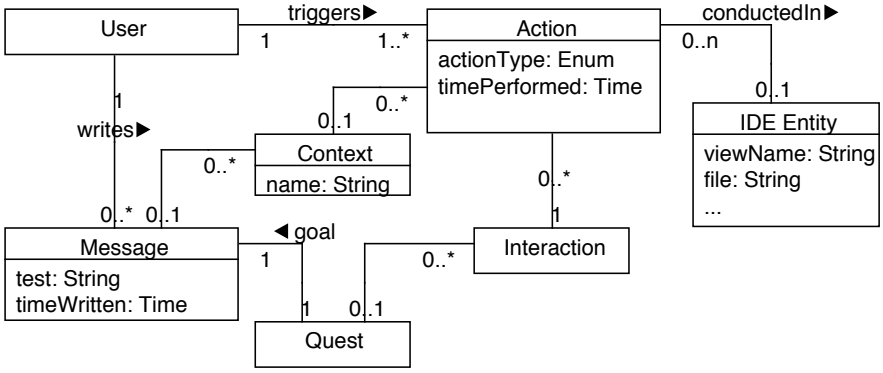


Figure 7.2: *Data model used in the repository*

7.2.1 Capturing IDE Interactions

We want to capture a fine granularity model of how developers interact with the IDE. Our minimal independent unit capturing user interaction within the environment (the IDE) is called *Action*. Actions refer to IDE features that can be executed by the user, such as opening a file, changing tab, selecting text, performing an editing operation, closing a file, and running a test case. As illustrated in Figure 7.2, for every detected action, we record the developer who performed it, the *IDE entity* involved (Java file *X*, Package Explorer view, *etc.*), the *type* of action (opening/closing of a view, editing, *etc.*) and the *date and time* at which the action has been performed.

Figure 7.3: *Example of user actions within the IDE on a timeline.*

Figure 7.3 shows a time line of actions a developer performs interacting with an IDE, while working on an ordinary task. On the time line we draw a vertical mark for every action detected, with more recent actions on the right. Actions are automatically collected and then processed. We group actions into *interactions*, according to their time proximity. Actions at a short time distance apart from each other will be part of a single interaction, modeling the fact that people take a few instants to decide on what to focus on. As an example, if a user closes a number of files one after the other (which is recorded as three distinct actions), we consider this a single interaction with the IDE (which would be

described as "closing files X, Y, Z"). Our heuristics is based on the time elapsed between one action and the next one. After the initial action, every other action in the same interaction has been performed within $x \leq \Delta t$ from the previous one. From observations during our initial experiments, we set $\Delta t = 3$ *seconds*.

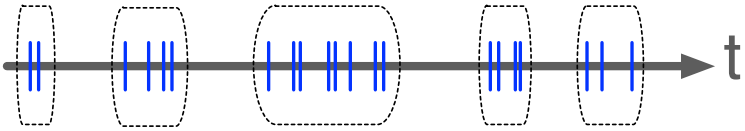


Figure 7.4: *Example of 5 user interaction with the IDE on a timeline.*

As an example, Figure 7.4 visually depicts the grouping the actions previously presented into five interactions. We can also notice that single interactions can differ from each other by various factors and degrees. Some interactions group few actions, while some others are bigger, grouping more actions. Moreover the distance between one action and the other in the same set can vary, however it is never greater than $\Delta t = 3s$.

7.2.2 Micro-blogging within the IDE

Users are requested to explicitly tell what they are doing in the form of a short, Twitter-like, message. Developers are encouraged to contribute in first person, discussing the things they care about in their code. For every message, we record the developer who wrote it and the date and time at which the message has been written. To encourage developers to keep their messages short, we propose a (Twitter-like) message length indicator, suggesting a maximum message length of 140 characters.

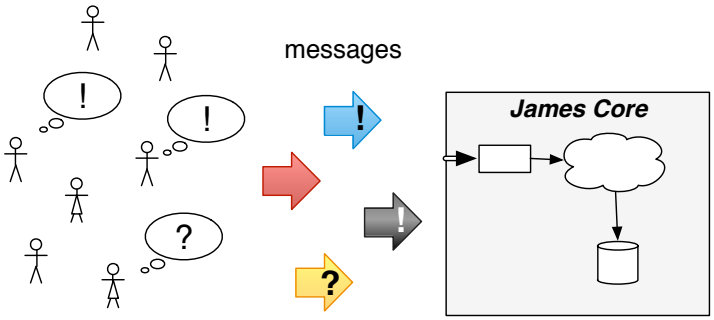


Figure 7.5: *Developers sending micro-blog messages*

Figure 7.5 depicts the micro-blogging scenario with a team of developers. Developers send a series of short short messages, in which they can express questions, remarks or any other information related to the software project. Messages are collected and then analyzed and stored into a central database. The analysis of messages includes their identification into categories (*i.e.,* questions vs answers) and the identification of keywords and concepts from the message.

7.2.3 Quests: Building a Knowledge Base

In our approach we combine a micro-blogging message and series of interactions into a *quest*. We refer to the message as the quest *goal*, and the interactions as the quest *trace*. Messages and interactions are furthermore connected by the *context*: an identification of the current project or maintenance task the developer is working on.

Figure 7.6 depicts how quests act as "containers" for a series of interactions. Micro-blogging messages are shown as taller lines with respect to actions, while the domain of a quest is represented as a rectangle.

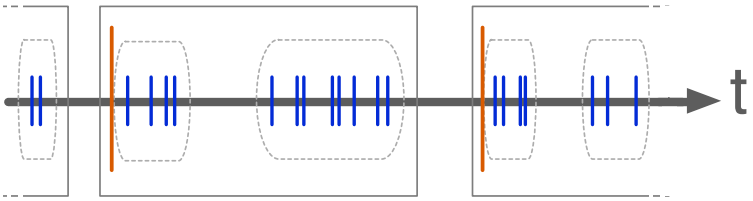


Figure 7.6: *An example of quests on a timeline.*

7.3 Implementation

We implemented the proposed approach in a tool named James. James follows the client-server architecture displayed in Figure 7.7. Our current implementation provides an Eclipse client, in the form of the James plugin. In the future we anticipate clients for, *e.g.,* IBM Jazz, Microsoft Visual Studio, and a fully web-based client. Note that our approach is language-independent, and thus can be applied to any IDE.

The Eclipse James plugin currently simply allows users to update their quest goal and collects navigation information through the use of listeners. The Eclipse view provided for entering messages and for following the messages sent so far is displayed in Figure 7.8. Micro-blogging messages and interactions are sent to the server to be analyzed and stored.

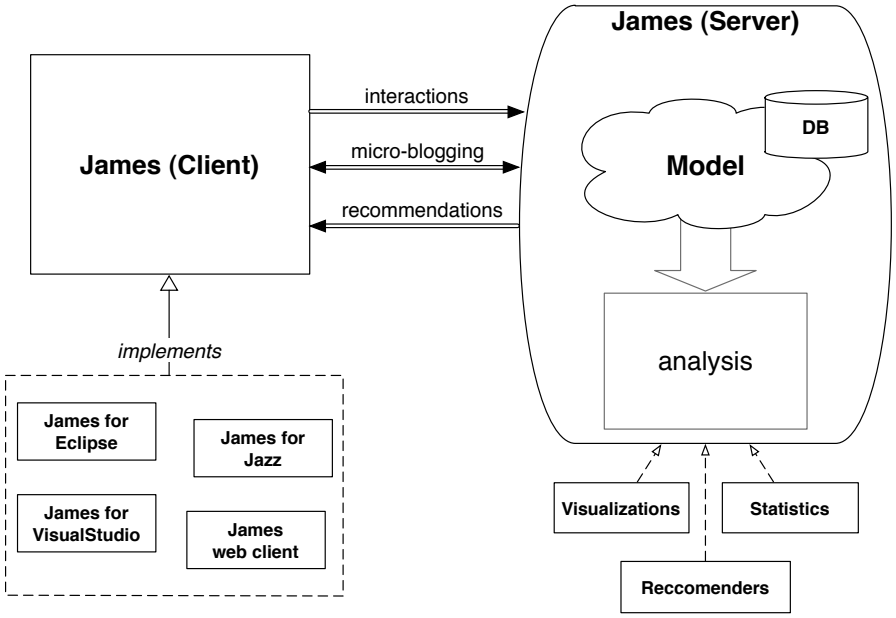


Figure 7.7: *Architecture of James*

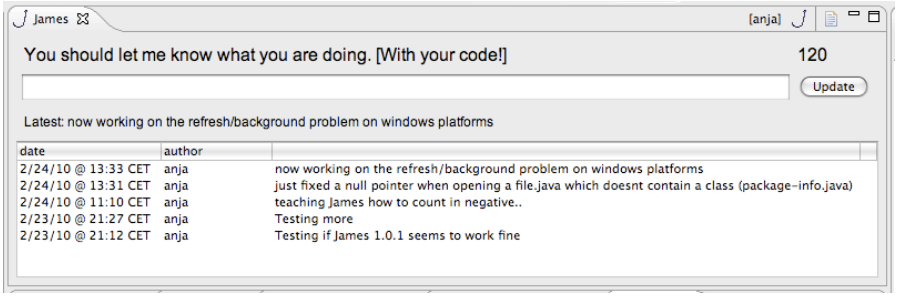


Figure 7.8: *James Client for Eclipse*

The James server provides a (MySQL) database and functionality for analyzing and combining messages and interactions from multiple developers using a James client (*i.e.,* the James Eclipse plugin). The schema adopted is based on Figure 7.2. The current prototype is primarily intended for gaining experience with the type of messages developers are willing and able to send, and how these relate to actual interactions. With the results of the present chapter in place, our next project will be to extend James with functionality for, *e.g.,* recommendations and visualizations based on the collected information.

7.4 Initial Evaluation

We conducted an initial evaluation with 7 developers working on 5 different projects. The goal of this explorative study was to provide first insights into the following research questions: (1) How often do developers change quest? (2) How many interactions are there per quest? (3) What do developers write in messages? and (4) How can quests support programming activities in multi-developer projects?

In the analysis of the gathered data, we focused on understanding whether and to which degree our approach has a valid foundation. We manually analyzed quest messages from all the involved developers to evaluate their willingness to share messages and we categorize messages according to information they provide. We first present statistical information, such as frequency and length, on the collected messages and interactions. Secondly, we report on initial findings from our manual examination of quests. Furthermore, we exemplify the potential benefit of the link between messages and interaction traces.

7.4.1 Study Setup

We distinguish two group of users: group 1 (*G1*) used James while performing their typical working activities (both in academic and industrial settings), while group 2 (*G2*) worked on a given task on a small sized (4,500 lines of code) Java system.

The set of software systems on which *G1* worked comprises:

- an **industrial** project, consisting of approximately 200,000 lines of code;
- **Crawljax**,[2] an open source tool for automatically crawling and testing Ajax web applications. The Crawljax core consists of approximately 19,000 lines of Java code;
- **JPacman**, an academic project consisting of 4,000 lines of Java code;
- an **academic** software project to profile plug-ins executions running in the Eclipse workspace.

Developers working on these projects used James during their typical activities during two weeks, and shared the database of collected messages and interactions with us.

2. http://crawljax.com/

The three developers in *G2* had to perform a given task on JAMES itself (4,500 lines of Java code). The task took approximately 4 hours. We provided a working version of the system where quest messages entered by users are directly shown in the view and then stored into a database. The given task was to "*Implement the retrieval of messages from the database. Messages from different users must be properly displayed in the* JAMES *view.*" We also provided a mock class with some comments as guideline. To implement the requested feature, knowledge about part of the system was needed (database connection, messaging, *etc.*). Every user had good high level knowledge of the underlying model and of the functionality provided by the artifact.

We deliberately chose JAMES as a artifact for our experiment. A very good knowledge of the underlying code was fundamental to asses the quality of the information provided by messages and interactions during the analysis of the collected data.

The Data Set

Table 7.1 reports on the total number of messages, interactions and actions collected for each user during the whole duration of the study.

user	project	# messages	# interactions	#actions
user 1	industrial	34	2,829	12,584
user 2	Crawljax	73	1,768	2,471
user 3	JPacman	66	478	763
user 4	academic	14	3,781	10,352
user *A*	James	36	381	572
user *B*	James	41	424	769
user *C*	James	36	221	319
7 users	5 projects	300	9,882	27,830

Table 7.1: *Data collected in our preliminary study*

We manually examined the data about users to identify *development sessions*. We consider a development session as a period of time when the developer is working in a continuous manner (*i.e.,* with only short breaks). To separate one session from another we looked at the time difference between user's messages and gaps between recorded interactions. To distinguish between development sessions as faithfully as possible, we used a threshold of 30 minutes, and manually checked quest messages and relative traces to better understand whether the session was finished. For *G1*, the group of users working on their daily activities, we identified a total of 29 sessions most of which counted between 3 and 9 messages. For *G2* there was a larger number of messages (35-40) in a single continuous development session. Distinguishing between development sessions gives a different (finer), granularity of details about consecutive quests, with respect to considering all the messages from one user as linearly consecutive. This has an impact, for example, on the analysis of the frequency of messages.

7.4.2 Data Analysis

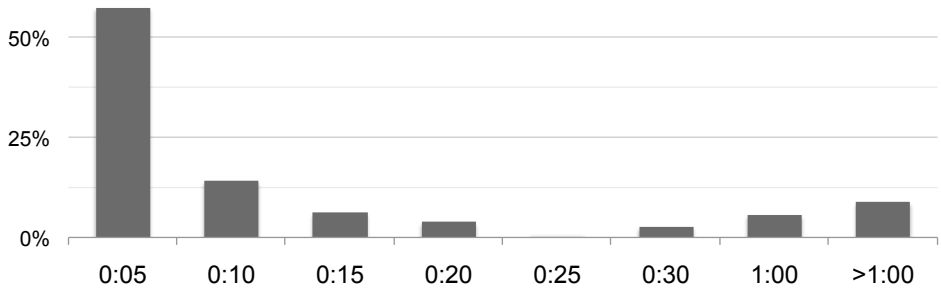


Figure 7.9: *Frequency of messages [hours]*

How often do developers change quest?

More than half of the messages have been set within 5 minutes from the previous one. This evidence is shown in Figure 7.9, where it is also possible to observe a particular trend in the distribution of messages frequency.

We can observe that messages that are not set within 5 minutes from the previous one, are likely to be set either within a short (10 minutes) or after a longer (1 hour or more) delay. Few messages or no messages have been set with distance of 20-30 minutes one from the other. Further inspection evidenced that such trend is common to both groups of users, thus both when performing ordinary maintenance work (either in an academic or industrial projects) and when participating in the experiment. This indicates that the frequency at which developers update their quest message is probably independent from the setting in which they work.

More details and confidence on the distribution of messages frequency is given by the box-plot in Figure 7.10: Half of the quest messages have actually been set between 45 seconds and 12:30 minutes after the previous one, with most of these being set after 3 minutes.

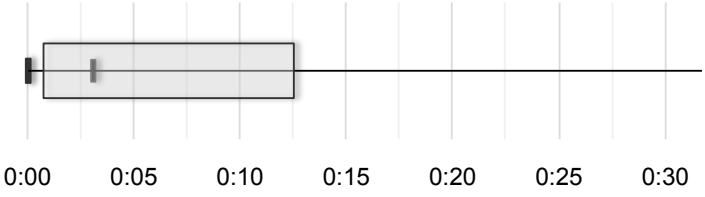


Figure 7.10: *Frequency of messages per session [hours] (upper value is 5:19)*

The number and frequency of collected messages are a first suggestion that users are willing to share what they are doing.

How many interactions are there per quest?

We collected a total of 9,882 interactions, grouping more than 27,000 actions performed by developers. Analyzing interactions in the context of the quest they belong to, we notice that most quests count a handful of interactions. Figure 7.11 depicts the frequency of the number of interactions per quest, taking into account all the collected quests. We can observe that more than half of the quests have less than 10 interactions associated to them: 2% of quests count one interaction, 13% count 2, 10% comprise 3 interactions. Of the remaining 75% quests, the 31% count between 4 and 10 interactions, while the remaining 43% more than 10.

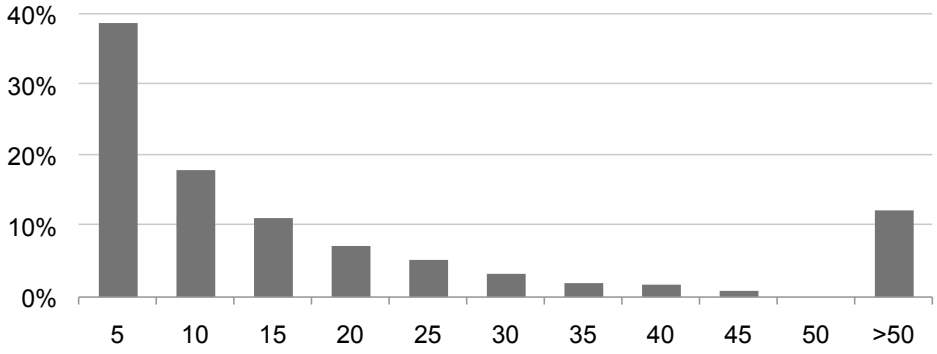


Figure 7.11: *Interactions per Quest [count]*

Considering the frequency at which developers updated their quest messages, we roughly collected 2.7 interaction per minute (8 interactions every 3 minutes). Furthermore, the large majority of interactions (66%) comprise only one action.We also analyzed the total number of actions in one quest. We observe that users exploit more than 10 features of the IDE (leading to actions) in one minute (see Figure 7.12).

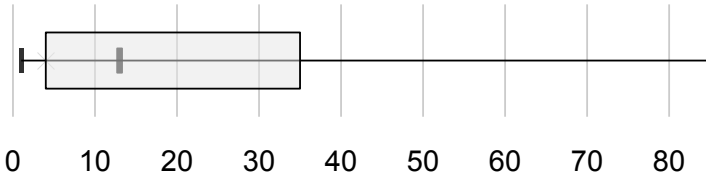


Figure 7.12: *Actions per Quest [count]*

What do developers write in messages?

Some users are more inclined to write a message as they are about to start a task, while others are more inclined to write a message in which they report the success/failure of what they have been doing in the last minutes. Furthermore, some messages were expressed in the form of questions. In a number of quests, developers are "talking to themselves" in the micro-blogging messages.

Categorization of messages

We notice that we can distinguish quest messages expressing activities (*i.e.,* what they are going to do and what they did) and messages commenting on (part of) the code. Some users also wrote to do's. We manually inspected the content of messages, categorizing them between messages expressing: intentions ("*Now I am going to...*"), ongoing activities ("*I am...*") and reports on a finished activity ("*I just did...*"), as well as comments ("*this is like so*") and to-do's ("*later I will need to...*"). Figure 7.13 visually describes the result we obtained from such categorization. We can see that only a minor part of messages does not fall into one of the proposed categories.

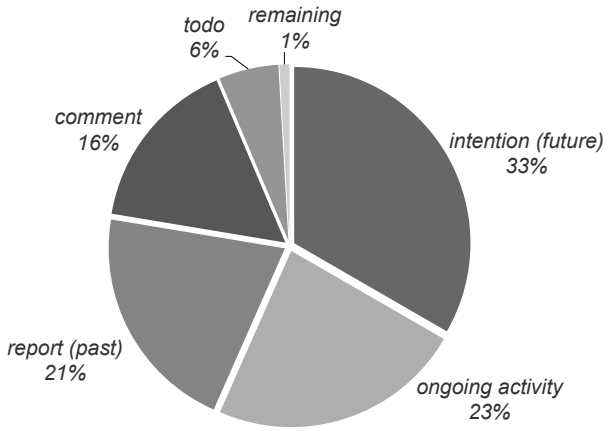


Figure 7.13: *(About) what do developers write in messages?*

We observe that 33% of the messages are about future, 21% report on past activity, while 22% cover ongoing activity, about which the message has been updated after the developer started working on her (sub)task. The remaining of the messages is divided among comments (16%), todo's (6%), and other sentences.

By inspecting messages very close to each other (within 30 seconds), we can notice that quest messages as close to each other as 30 seconds, are either directly correlated, with the second message acting as "annotation" for the previous quest message, or the first message states the end of the previous activity. An example of the first case are the following

messages: "*so let's check, whether the SQL query works*", together with: "*first figure out where the job is invoked ;-)*". While "*First run finished*" followed by "*now switching to CrawlQueue*" is an example of the latter.

Keywords

Some words occurring in messages, together with the verb tense used, have been determinant to establish in which category each message was falling into. We therefore tried to identify a set of such *keywords* recurring in messages from different users. It turns out that words such as *now*, *going (to)*, *test*, *seems*, *starting*, *checking*, and *trying* are recurring in many messages and among different users. Note that they can often be found directly at the start of messages. Figure 7.14 illustrates the 20 most frequent identified keywords in a word cloud[3], which also displays their frequency (as size of the words). Furthermore, two users explicitly expressed to-do's using an hash (#) in front of the keyword '*todo*', simulating hashtags in Twitter.

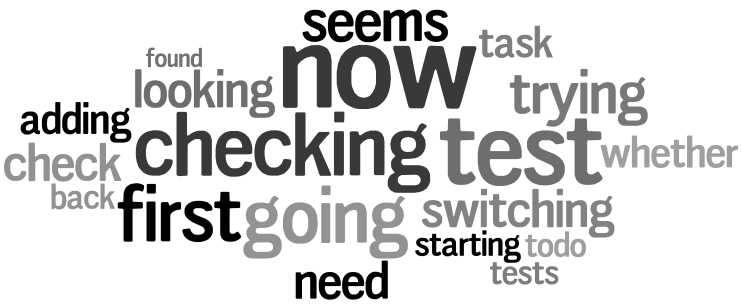


Figure 7.14: *What keywords can we find in messages?*

Content of Messages

Content of messages in a development session seem to be sufficient to have an idea of what the developer have been working on during the session. As an example, Figure 7.15 shows a word cloud with words (except keywords) from messages set by user *A*, who worked on the given task, implementing the ***retrieval*** of ***messages*** from a ***database***.

To give a better feeling about the content of quest messages set by developers, we present a collection of messages in our data set:

1. "*first figuring out how to connect to the server*"
2. "*testing to see the importance of the synchronized statecompartor*"
3. "*Investigating failing error in JETsGenerator*"
4. "*Trying to figure out how to create a proper UUID from an int in the database.*"

3. Images created by Wordle.net (`http://www.wordle.net/`).

Figure 7.15: *Words most frequent in messages by user A.*

5. "*No real significant differences found between CrawlQueue / SpeedQueue*"
6. "*Finding out that 'blue' actually means green here*"
7. "*seems that QuestMessageCapsule has all the info I need to get the right fields out of my query*"
8. "*MoveTest.testApply actually only tests moving towards an empty cell.*"
9. "*where is the code that is sending a message to the database?*"
10. "*Quests should be kept into a sorted list. #todo*"

Talking about code elements

We observe that a fair share (28%) of the collected messages mention a code element (package, class, method or attribute). Such messages mentioning code elements are distributed across all the message categories and that the portion of quest messages mentioning classes, methods, *etc.* is interestingly similar for every user. This fact, together with the similarity in the frequency of messages, further suggests that developers have similar habits when communicating (about) what they are doing.

Messages length

Regarding the length of messages, we observe that more than half of the messages (58%) have length between 20 and 80 characters, with an average of 54.5 character per message. On average, users wrote 8.6 words per message. This indicates that the limit suggested by James of 140 character per message may be sufficient to express what they are doing.

How can quests support programming activities in multi developer projects?

We manually analyzed interactions combined with a subset of the quests with messages mentioning code elements, in particular class names. Our finding is that most of those interactions involved navigation inside the mentioned class file, in particular, as we might expect, when the message expresses an intention. Furthermore, analyzing quests from

users in *G2*, we noticed that a portion of the relative messages refers to common problems faced by the developers.

We hypothesize that access to the knowledge base about the system could have helped (later) developers in their programming activity. To evaluate this hypothesis we inspected interactions associated quest goals with similar content from developers in *G1*. We report on 3 cases, in which messages could have supported program comprehension.

Case 1: Solution found by user *B* can be useful to user *C*.

User *B*: "*quickly check how to iterate over a ResultSet*"
User *C*: "*Looking for an example how to use a resultset.*"

Both users eventually inspected the same file before setting a new quest message. This gives an insight into the usefulness of interactions associated to messages, at least for messages stating intentions and ongoing activities.

Case 2: User *C* has the information user *A* is looking for.

User *C*: "*How do I turn a timestamp from SQL into a Java timestamp?*"
User *A*: "*need to check online on how to include operations on the timestamps in the query (i.e., ≥)*"

Interactions from user *A* does not include navigation to any class file in the project. Her following quest messages indicates that she found the wanted information after 10 minutes. On the other hand, user *C* has been inspecting the code: In particular he has been browsing classes relative to the quest object representation, eventually terminating his journey on the class which contains an example SQL query involving a time stamp. User *C* sets a new message after 10 minutes, which suggests the issue was solved. The Knowledge gained by user *C* during his quest could have been "reused" by user *A*.

Case 3: Message from user *A* is the solution to the struggling of user *B*.

User *A*: "*I think startPlugin() and stopPlugin() are good places to start/stop the job.*" (Q5)
User *B*: "*first figure out where the job is invoked ;-)*" (Q6)
User *B*: "*postponed starting - have to figure out first where to start the job*" (Q7, 16 minutes after Q6)

We can see that the answer to Q6 is directly embedded in Q5. However, not having access to this information, user *B* spent quite some time browsing class files in the project before eventually reaching the same conclusion than user *A* (2 minutes after setting Q7). Almost 20 minutes could have been saved to user *B* by having access to the quest message previously expressed in Q5 by user *A*.

7.5 Discussion

7.5.1 Summary of Findings

The number, frequency and content of collected messages indicate that developers are willing and inclined to share what they are doing by means of a short micro-blogging message, regardless of the setting in which they work. Developers perform approximately 10 actions per minute, which are grouped into 2-3 interactions. A new quest goal is set, in most cases, every 5 minutes.

We categorized the content of messages into 5 categories and observed that one third of them expresses future intentions. Messages referring to concluded and ongoing activities account for one fifth each. Remaining messages include comments and todo's. Roughly one third among all the collected quest contain an explicit reference to a code element (*i.e.*, a class name) in their message.

We analyzed how messages connect to interactions and we try to assess whether these connections are in principle meaningful. By manually comparing quest goals expressed in similar messages, we observe that quests provide information that is potentially meaningful to different developers working on similar tasks, both in the associated traces and in the goal themselves.

From the conducted exploratory evaluation, we can conclude that knowledge about the software being changed, constantly built up by developers, can be captured in the form of quests. Accessibility to this knowledge base can support developers during maintenance.

7.5.2 Interpretation of Findings

Impact of message sharing on message frequency

Developers participating to our experiment updated their quest message once every three minutes. This research motivates the usefulness of sharing such (short) messages with other developers within the same team. We wonder and plan to study to what extent the sharing of messages between developers impacts the messages frequency.

We hypothesize that the frequency of messages would slightly decrease because a developer might "filter out" those messages in which she mainly "talks to herself." We will evaluate this hypothesis by comparing the finding reported in this chapter with those obtained by the analysis of messages from future experiments conducted in a similar setting that the one involving group *G2*. We will try to quantify the variation in the frequency of messages when developers in the same team (1) can see each others messages and (2) receive recommendations based on their current quest goal.

Interaction Resolution

Our Eclipse James plugin captures interactions by modeling navigation information in the IDE, such as browsing through projects files. However, other programming activities, such as writing code, are not currently monitored. From the conducted study, we observed that with a timer on interactions of 3 seconds, two thirds of the collected interactions are limited to one action. Since our heuristics groups actions into interactions based on the time elapsed between one action and the next one, we estimate that modeling changes to the source code as interaction activities in the IDE, would significantly affect traces associated to quests involving writing code, whereas traces relative to quests with a program comprehension activity as goal would resemble those currently collected by James.

We plan to further investigate our current algorithm to clustering actions into interactions, as well as alternatives. We intend to both extensively analyze the collected actions and to monitor other developer activities (such as editing or browsing code) in order to determine a better approach into building interaction traces.

Code completion in messages

Developers referring to code elements in quest messages and the correlation we encountered between mentioned class files and browsed class files suggest that James should provide support to developers, for example in the form of auto-completion program elements such as names of packages, classes, methods and attributes. Such support will be convenient for the users and in particular way useful during the analysis of messages. Such a feature will not only avoid spelling mistakes, but identify program elements as such, establishing a direct link between the messages and the code itself.

Connecting messages to interactions

We obtained promising results from the first analysis of traces correlated to quest goals about future intentions and we are confident that associating interactions to micro-blogging messages have great potential. While our heuristics associating to a quest all the interaction between the setting of its goal and the next one seems to be valid for such quests expressing intentions, we need to evaluate its quality in regards of quests with messages falling in the other categories.

As future work, we will research how to improve the heuristics to attribute the appropriate interactions to a quest. We hypothesize that categorization of messages can help refining our heuristics to determine which interactions are to be associated with a quest. For example, when the quest goal reports about the completion of a task, it is likely that at least part of the interactions preceding it are linked with it. We will try to evaluate whether and to which extent categorizations of messages can help determining an appropriate association between quests and interactions.

Messaging conventions such as hashtags and emoticons

We observed that in some of the collected messages users used the hash symbol in front of the *todo* keyword, forming an *hashtags* (*#todo*), as Twitter. We also noticed that in some of the messages users try to express their feeling with (informal) onomatopoeic words such as *yes!*, *mmh*, *grr*, *oops*, and *yu-uh*. Users also included emoticons (*e.g.,* :-)) in their messages. We believe those information could be valuable to understand the value of the quests they belong to. For example, a message such as "*Messages are nicely stored in the fresh mysql database :-)*" leaks out that the user is happy with the solution he found/implemented.

We intend to further investigate the use of (hash)tags and other methods people use to enrich their micro-blog messages and whether we can benefit from these findings, for example for a more detailed or different categorization of messages.

Furthermore, we envision that in addition to actively indicate quest goals, in our approach, users will have the possibility *tell* the IDE whether their journey through the code (the trace) has been successful and the quest goal is accomplished. By capturing the "enlightenment moment" when pursuing a quest goal, additional value is added to the user's navigation (since it means that the quest has been accomplished while following those particular steps). This information is particularly important when sharing this knowledge with other developers.

7.5.3 Applications of James Data

Our purpose is to build a shared knowledge base with information from developers, which captures information concerning developers' knowledge. Such knowledge base can then be used to support development activities. We propose two applications: (1) sharing micro-blog messages to increase members awareness and (2) sharing navigation data in the form of (targeted) recommendations to improve software comprehension.

Increasing Knowledge Awareness

Micro-blogging is an asynchronous approach on information exchanging. However, such (short) messages are often seen by other users in a semi-synchronous fashion (usually within the day or within a few days at most). James takes care of transmitting the messages to interested users. With *interested users* we mean both users that actively want to "follow" (in a Twitter-like manner) another user, a project, a file, a concept, etc. and also users for which the message can be relevant. We can determine the relevance of a message for a user, given his previous messages and navigation data. James will point out related messages, when this is relevant to improve the user's work. For example, it will direct the user to another developer who previously faced a same or similar concern than expressed in her message. In this way the users can actively take advantage of knowledge of others(by contacting other users to directly ask them about their findings). Establishing in

this way a foundation for *collaborative program comprehension*. We can also have sufficient information to figure out when messages are obsolete. For example, if they occur in a part of the code that has later been deleted, or if a message mentions an identifier that has changed.

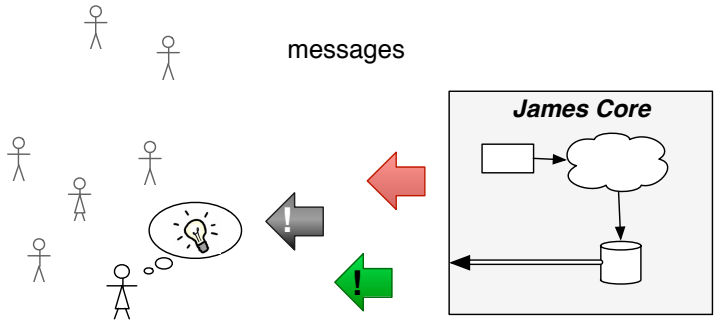


Figure 7.16: *Developer receiving micro-blog messages*

Figure 7.16 depicts the sharing of micro-blogging messages, previously collected within the development team. A user is notified with a subset of the messages previously collected and analyzed by James. Messages are redirected only to those users, for which they can be relevant (*e.g.*, to solve the concerns they expressed in their quest message).

Recommending Comprehension Paths

By combining quest messages and navigation data, we can recommend to users where to look in the code, given their quest goal (this can be done when we captured how others solved the same or a similar issue). The information gathered is processed (*interaction traces* are built), and then used to build a general knowledge about the system, which can be used for a number of applications. For example, James can suggest where to look in the code, given a developer's goal and his/her private navigation history, as shown in Figure 7.17. The collected knowledge about the system is filtered by James, so that only relevant information is reported to the individual engineer.

Among the challenges to be addressed when building such a recommendation system, one important issue is finding meaningful criteria for the identification of information relevant to the user's goal (as opposed to noise generated by browsing irrelevant parts of the system). Once the information is filtered, it needs to be merged with the previously recorded data (collected from many users). The merging of the data gathered from different users is another point of investigation (for example, traces from different users

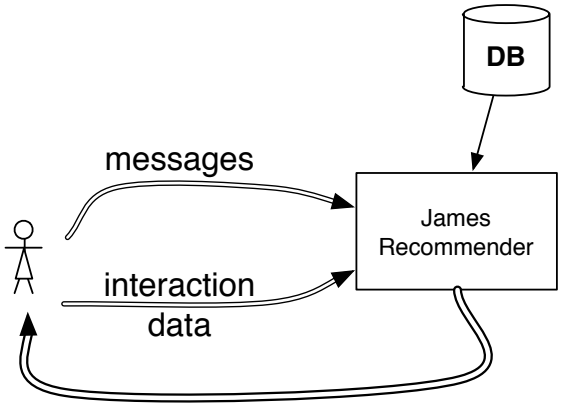


Figure 7.17: James *recommendation approach schema*

working toward the same goal can have a different importance, based on some yet to be defined metrics, such as developer experience on the code and shorter path to solution) as well as the scalability of the tool due to the possibly huge amount of data collected.

Another important research question to be tackled is how the collected data can survive code refactoring, while maintaining its valuable information.

7.6 Related Work

Our research builds upon several (software engineering) disciplines. First, there is related work concerning studies of what developers do, and what information they need from the IDE. As an example, Sillito *et al.* provide a study of questions asked during programming change tasks [161], and Ko *et al.* report on an etnographic study of how developers work at Microsoft and what their information needs are [103].

Software development and maintenance are inherently collaborative activities: A survey of research in the area of collaborative software engineering is provided by Whitehead [176]. Web 2.0 provides new ways of collaboration and informal communication [131], and the incorporation of Web 2.0 techniques in software development is attracting more and more attention both in industry and academia [169, 171]. As an example, IBM's Jazz[4] incorporates the possibility of adding tags to work items, and its use by IBM developers has been studied extensively by Treude and Storey [168]. Another example of the integration of Web 2.0 into the software development process is the work by Begel *et al.* on Codebook (inspired by Facebook) to integrate several repositories relevant to software development [11] and by DeLine on bookmarking in code (inspired by Delicious) [46]. Micro-blogging is an important element of Web 2.0, and thanks to the massive success

4. http://jazz.net/projects/content/project/plans/jia-overview/

of, *e.g.,* Twitter, an active area of research itself[5] [65]. We are not aware of other research studying the potential of micro-blogging during software development. Similar to some extent to micro-blogging, however, are Internet Relay Chat (IRC) discussions, and their use during the development of the Linux Gnome code has recently been analyzed by Shihab [160].

A number of existing studies report on the meaningfulness of navigation traces and their potential. Fritz *et al.* conducted an empirical study assessing the relationship between programmer's activity and what a programmer knows about a code base [60] and DeLine *et al.* report results of two studies which demonstrate that sharing navigation data can improve program comprehension and "is subjectively preferred by users" [47]. Both Mylyn [101] and NavTracks [163] are navigation aids based on what the programmer is currently looking at in the IDE, to recommend other entities to look at. Additionally, a study by Robbes on recommender systems based on recorded interactions [146] recognizes the lack of support for interaction annotations.

7.7 Concluding Remarks

During the process of trying to understand a piece of code, developers build up a substantial body of knowledge on the code they are inspecting—knowledge that often evaporates after the corresponding maintenance task is finished. In this chapter, we propose a method to capture this valuable information, by recording how developers interact with the code, and by encouraging developers to tell their team members what they are doing.

The key contributions of this chapter are the following:

- A novel method for recording program comprehension knowledge by combining micro-blogs expressed by developers with interaction data collected by the IDE;
- A client-server implementation of this approach by means of the James Eclipse plugin;
- An empirical evaluation of the proposed approach, giving initial evidence that developers are willing to micro-blog on their activities, and that the combined interaction and micro-blogging data is helpful in subsequent maintenance tasks.

Based on our first experiments, we consider the combination of micro-blogging data and automatically collected interaction data a highly promising route for recording and sharing knowledge built up in the program comprehension process. Future research directions include enriching the tool suite with additional mechanisms such as providing the ability to follow specific developers, projects, or work products, and enhance quest visualizations. Future research directions also includes carrying out larger scale case studies in which teams will be using James for a longer period of time. This will help investigating, for example, the impact on James of frequent interruptions of developers' work [103] and multitasking, which can break a logical task into many sessions separated in time.

5. See also the bibliography at `http://www.danah.org/researchBibs/twitter.html`

Part IV

Finale

8 Conclusion

8.1 Contributions

The goal of this thesis is to provide additional effective support to developer's teamwork, by devising lightweight IDE additions that can be seamlessly integrated in the development workflow of a variety of teams (*e.g.,* from small and co-located to large and distributed) and that require little learning time. To that end, we investigate how to support developers' teamwork in the IDE, implement the approaches emerged from our investigations as lightweight and unobtrusive extensions, and test the approaches with users in different studies. In this dissertation, we make the following key contributions:

In Chapter 2, we present an in-depth investigation, conducted through qualitative analysis, of developers' current teamwork needs. This results in:

- additional evidence that teamwork needs mostly regard coordination aspects, corroborating previous research;
- novel evidence that developers are able to face scenarios considered problematic in literature;
- evidence that developers find it hard to deal with breaking changes, but they get frustrated only if the breaker is internal to the project;
- recommendations on how to improve collaboration in teamwork in the software implementation phase.

In Chapter 3, we present an in-depth qualitative evaluation of the role of development mailing list for OSS project communication. This results in:

- a coding system that is reusable for analysis of developer communication in general, and mailing lists in particular;
- the assessment of relative frequency of topics in developer mailing lists;
- the assessment of relative participation of developers in developer mailing lists;

- the implications for researchers and practitioners, derived from our findings about mailing list communication in OSS systems.

In Chapter 4, we conduct an in-depth analysis, based on interviews and surveys, to understand how and why enterprise software developers communicate with one another, and how often they do so; then we propose an IDE extension to support the person discovery, selection, and communication process. This results in:

- the criteria developers use to identify and choose a set of relevant people, how they select the best person to contact, the means by which they contact that person, and how often their conversations led to positive working relationships;
- CARES, a fully implemented IDE extension to support the person discovery, selection, and communication process;
- the evaluation of CARES with professional developers in two studies spanning up to four months, which reveals that the reaction to CARES by most developers has been primarily positive;
- evidence that that the deployability of a proposed IDE extension is strongly influenced by ease of installation, simplicity of use, and effectiveness at a single task.

In Chapter 5, we present an analysis of the current support for receiving code changes in the IDE, based on widespread usability heuristics, and the subsequent design of an IDE extension to improve teamwork support. This results in:

- a set of requirements for a tool to support teamwork based on the unmet usability heuristics;
- the design of BELLEVUE, an IDE extension to support teamwork by improving the integration of code changes in the IDE;
- the evaluation of BELLEVUE, conducted by involving nine professional developers, which reveals that developers' reaction to the design was very positive.

In Chapter 6, we present the results of an online survey and several interviews conducted with professional software engineers on current usage of code bookmarks and the subsequent design of an approach to code bookmarking. This results in:

- data on current usage of code bookmarks;
- a set of requirements for a non-intrusive bookmarking tool that facilitates information sharing;
- POLLICINO, a novel approach to code bookmarking, which we designed and fully implemented as an IDE extension;
- the evaluation of POLLICINO and the potential of collective code bookmarks, in an exploratory pre-experimental user study with eleven participants.

In Chapter 7, we present an approach for creating information during program comprehension, which is useful in later consultation and in teamwork scenarios. This results in:

- a novel approach based on the idea of combining micro-blogs expressed by developers with interaction data collected by the IDE;
- James, a client-server implementation of the aforementioned approach as an IDE extension for Eclipse;
- the empirical evaluation of James, and the initial evidence that developers find it reasonable to micro-blog on their activities, and that the combined interaction and micro-blogging data help sub-sequent maintenance tasks.

8.2 Reflection on the Research Questions

Taken together, the contributions serve to answer the research questions that we set out to answer in the introduction. Here, we reflect on these questions, trying answer them in the light of our findings.

[1] How do developers experience collaboration in teamwork?

To better understand the leeway for improving teamwork support in the IDE, we first investigated the current practice of developers' working in teams (presented in Chapter 2). To our surprise, the interviewed professional developers reported to use effective workarounds that make them able to deal with different scenarios considered as problematic in literature (*e.g.,* inefficient task assignment). Nevertheless, they find it still difficult and time-consuming to correctly understand code changes made by other people (*e.g.,* for maintenance tasks or for finding the source of an unexpected error); moreover, they find this particularly frustrating when these changes are done by somebody working on the same project, because of the lack of coordination effort by the author of the changes (who, for instance, could have shared additional information with them).

Firstly, these findings made us realize once again the importance of verifying whether assumptions generally accepted in previous research also hold in the specific context under investigation; finding that some scenarios were not seen as problematic allowed us to concentrate on more relevant issues. Secondly, the findings showed that, although developers' questions might be answered by tools already available from research, these tools are not used by the developers; this underlines the importance of more research in this area, with the specific aim of creating solutions that can be easily adopted in different working scenarios, without requiring major changes or expensive learning. Finally, the findings reiterated on the importance of sharing information to support teamwork; when information is necessary, but not visible or communicated in advance, developers have to spend a significant amount of time to look for it, thus leading to inefficiencies and sometimes even frustration.

In the following question, we move from the industrial context to the open source one.

[2] How is information shared in open source software projects?

To understand how information is shared in OSS systems, we conducted an in-depth analysis of the communication channel that is considered the hub of project communication in OSS projects: the development mailing list. This study helped us to better understand the type of information that is exchanged by open source developers, so that we can guide the subsequent efforts to the most productive directions. In our case study, we found that communication is presently scattered across several types of channels, such as issue repositories, code commits, face-to-face talk, and online chats. Interestingly, the mailing list seems to have lost its predominant role in favor of the issue repository, which is closer to development artifacts (*e.g.,* source code entities) and allows a more focused and structured communication than noisy emails. This is evidence of the importance of creating ways to improve intra-team communication that are linked to the development process, as how we tried to do with our proposed approaches.

Moreover, an amount (larger than we expected) of OSS mailing list communication is devoted to less technical details and more social and organizational issues. This has to be taken into account in future endeavors to support teamwork with tools.

Considering the lessons learned from answering the first two questions, we focus on ways to display already-recorded information that is useful to support teamwork, and on ways to generate additional information that is currently not recorded but that would be useful to teamwork. These are the pillars that we investigated in the following research questions.

[3] How can we expose existing information to support teamwork?

To better understand the type of information that would be useful to developers working in teams, in addition to the studies presented in Chapters 2 and 3, we conducted interviews and surveys as presented in Chapter 4. We underline that much of the information that would help developers is already recorded and available, for example, in the code change history. In fact, questions such as "Who is an expert of this piece of code?" can be answered by simple queries to the versioning system. Nevertheless, the interviewed and surveyed developers did not seem to profit from this information. This is not surprising, because "the ease of acquiring information is at least as important as the quality of the information in determining the sources that people use" [105]. We focus our answer to this question on finding how to make this historical and team information easily accessible.

We proposed two approaches, in the form of IDE extensions, to expose information that is already available, but not easily accessible in the IDE. CARES (Chapter 4) is our first addition: It gives information about who one should contact to ask information on a specific piece of code. A clean interface that shows contacts' information, their availability, and their photo proved to be effective and useful to developers. BELLEVUE (Chapter 5), our second addition, improves the code change support offered by the IDE: Not only does it keep visible what changed since the last update of the local code, also it seamlessly integrates code history in the editor and adds contact information similarly to CARES.

Three points that emerged from our findings are particularly interesting: First, lightweight solutions, which require neither huge implementation effort nor long learning time, could reach the desired effect of supporting teamwork in the IDE. Second, developers' photos, which enable group members to easily recognize one another, increase their sense of community. Third, despite several usability problems with the current support for dealing with code changes in the IDE, popular programming environments have neither acknowledged nor solved them. This might be due to the fact that "the identification of specific, potential problems in a human-computer dialogue design is difficult" [122]. In fact, we only became aware of these usability issues after long investigation. We hope that our efforts will inspire similar industrial and open source solutions in this area.

[4] How can we aid information creation to support teamwork?

Not all the information that would be useful to developers is already recorded. For example, when conducting program comprehension tasks, developers do not leave traces in the source code management system, because they do not make modifications to files. For this reason, all the precious time developers spend in understanding some parts of the system is often lost once a task is completed. We argued that this kind of information would be very valuable to support teamwork. In Chapters 6 and 7, we focused on this instance of creating information to support teamwork. We addressed our research question by investigating two approaches that support developers in sharing the information they collect while spending time on program comprehension tasks.

The study on Pollicino focused on code bookmarking. We found that the existing support in the IDE is currently underused and we proposed a simple, lightweight solution that showed how such an approach can be effective to generate information that is useful for other team members afterwards. The study on James made a step further and proposed an approach that let developers share their comments in form of brief messages and automatically records their interactions with the IDE. An initial study with developers demonstrated the feasibility and the interest of such an approach.

Overall, we showed that creating and recording additional information to better support teamwork do not require overly complicated approaches, which might disrupt developers' productivity and waste their time. With simple, short textual notes developers can share information valuable to their peers. Participants to our studies showed interest in such lightweight approaches.

8.3 Future Work

The future steps that we envision as a natural continuation of the presented work are numerous and are mostly discussed in each chapter, in the specific context from which they are emerging. Nevertheless, from a higher level perspective, we foresee few directions that we introduce here.

People in the IDE. The study on CARES especially reported that the small detail of the photo of the people to contact made a great difference in developers' perception of our solution. In our opinion, this underlines that people are an undervalued part of the development process and that we should find ways to better integrate information about people in the programming environment. As a future work we foresee extensions that include people as first-class entities in the IDE, together with source code artifacts and the development process.

Moreover, the presence of photos increased the sense of belonging of team members, probably indicating that the distance perceived among developers was reduced. In this context, Dullemond *et al.* investigated the use of mood indicators within a microblogging solution for developers [53]. They found that the mood associated with microblogging would made people feel more connected on a social level. An interesting venue for future work would be to include mood-sharing capabilities and photos of the posting developers to our JAMES extension. A quantitative evaluation can then be conducted both on the effects on the development process and outcome and on whether the perceived distance changes, using the model proposed by Prikladnicki [140].

Integration within the development process. Our research on BELLEVUE is a first step in the direction of improving the development process by make it more integrated with the development product. We found that the current support for managing code changes in the IDE is very basic and suboptimal. We proposed a solution to this problem and we envision a more comprehensive approach that considers all the different facets of the development process (*e.g.,* bug management and new feature requests) and integrates them seamlessly in the IDE. On this, we support solutions that would provide progressive and unobtrusive disclosure of useful information, rather than heavyweight solutions interfering with programmers' productivity or forcing developers to change their working style.

Moreover, although we aim at supporting teams independently from the development process they have in place, an interesting venue for future research would be to test in which situations our solutions work more effectively or achieve better results. Agile methods improve the sense of responsibility and belonging of team members, thus leading people to be willing share more [117]. In this context, our solutions to help developers creating and sharing information to support teamwork could be more effective; interesting venue for future work would be to investigate this hypothesis empirically.

Generalization. The studies we conducted are based on qualitative methods [55]. This allowed us to obtain data grounded in the experience of the participants, rather than solely in our speculations or in previous literature. Moreover, we obtained rich information on what people found useful or lacking in our solutions. However, mainly because of the (time-consuming) nature of qualitative methods, this usually means having less data points to generalize our findings. Although sometimes we used surveys to tackle this aspect (Chapter 4 and Chapter 6), in general, we

could only draw initial conclusions, for example, on the usefulness and effectiveness of our tools. An important venue for future research would be to check the findings of our exploratory investigations with a larger pool of participants (Chapter 2 and Chapter 7) and with more systems (Chapter 3); another would be to conduct quantitative measurements on the impact of our proposed tools and designs, for example with a longitudinal study in the daily practice [148] or with controlled experiments [104].

Bibliography

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of ICSE 2006 (28th ACM/IEEE International Conference on Software Engineering)*, pages 361–370. ACM Press, 2006. (Cited on page 20.)

[2] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996. (Cited on page 21.)

[3] Earl Babbie. *The practice of social research*. Wadsworth Belmont, 11th edition, 2007. (Cited on page 106.)

[4] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone? In *Proceedings of FASE 2010 (13th International Conference on Fundamental Approaches to Software Engineering)*, pages 59–73, 2010. (Cited on page 33.)

[5] Alberto Bacchelli, Michele Lanza, and Marco D'Ambros. Miler: A toolset for exploring email data. In *Proceedings of ICSE 2011 (33rd ACM/IEEE International Conference on Software Engineering)*, pages 1025–1027, 2011. (Cited on page 36.)

[6] Alberto Bacchelli, Michele Lanza, and Vitezslav Humpa. RTFM (Read The Factual Mails) –augmenting program comprehension with REmail. In *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)*, pages 15–24, 2011. (Cited on page 94.)

[7] Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg, editors. *Human-computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers Inc., 1995. (Cited on page 4.)

[8] Aaron Bangor, Philip Kortum, and James Miller. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, July 2008. (Cited on page 93.)

[9] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4(3):114–123, May 2009. (Cited on page 93.)

[10] Olga Baysal and Andrew J. Malton. Correlating social interactions to release history during software evolution. In *Proceedings of MSR 2007 (International Workshop on Mining Software Repositories)*, page 7. IEEE Computer Society, 2007. (Cited on page 33.)

[11] Andrew Begel and Robert DeLine. Codebook: Social networking over code. In *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*, pages 263–266. IEEE Computer Society, 2009. (Cited on page 138.)

[12] Andrew Begel and Anja Guzzi. Graphical user interface for integrated development environment tool, May 2, 2013. US Patent App. 13/282,415, US Patent Pub. US20130111428 A1, `http://www.google.com/patents/US20130111428`. (Cited on page 62.)

[13] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 125–134. ACM, 2010. (Cited on pages 16, 56, 57, and 68.)

[14] Andrew Begel, Nachiappan Nagappan, Christopher Poile, and Lucas Layman. Coordination in large-scale software teams. In *Proceedings of the CHASE 2009 (2nd International Workshop on Cooperative and Human Aspects of Software Engineering)*, pages 1–7. IEEE Computer Society, 2009. (Cited on pages 4, 26, 27, and 28.)

[15] Nicolas Bettenburg, Emad Shihab, and Ahmed E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proceedings of ICSM 2009 (25th IEEE International Conference on Software Maintenance)*, pages 539 –542. IEEE Computer Society, 2009. (Cited on pages 34 and 36.)

[16] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems*, pages 1313–1322. ACM, 2007. (Cited on page 27.)

[17] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of MSR 2006 (International Workshop on Mining Software Repositories)*, pages 137–143. ACM, 2006. (Cited on pages 32, 33, and 38.)

[18] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of FSE 2008 (16th ACM International Symposium on Foundations of Software Engineering*, pages 24–35. ACM, 2008. (Cited on pages 32, 33, 38, 40, 43, 48, and 49.)

[19] Sue Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263–, September 2001. (Cited on page 21.)

[20] Gregory Alan Bolcer and Richard N. Taylor. Endeavors: a process system integration infrastructure. In *Proceedings of ICSP 1996 (4th International Conference on Software Process)*, pages 76–89, 1996. (Cited on page 4.)

[21] John Brooke. SUS: A 'quick and dirty' usability scale. In Patrick W. Jordan, B. Thomas, Ian Lyall McClelland, and Bernard Weerdmeester, editors, *Usability Evaluation in Industry*, chapter 21, pages 189–194. CRC Press, 1996. (Cited on page 93.)

[22] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983. (Cited on page 4.)

[23] Andrea Brühlmann, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Enriching reverse engineering with annotations. In *Proceedings of MoDELS 2008 (11th International Conference on Model Driven Engineering Languages and Systems)*, pages 660–674. Springer-Verlag, 2008. (Cited on page 101.)

[24] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, 2013. (Cited on pages 6 and 20.)

[25] Donald Campbell and Julian Stanley. *Experimental and quasi-experimental designs for research*. Rand McNally, 1963. (Cited on page 106.)

[26] Erran Carmel and Ritu Agarwal. Tactical approaches for alleviating distance in global software development. *IEEE Software*, 18(2):22–29, March 2001. (Cited on pages 4 and 6.)

[27] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of ESEM 2008 (2dn ACM/IEEE International Symposium on Empirical Software Engineering and Measurement)*, pages 2–11. ACM, 2008. (Cited on page 4.)

[28] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, November 2009. (Cited on page 25.)

[29] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of CSCW 2006 (20th ACM Conference on Computer Supported Cooperative Work)*, pages 353–362. ACM, 2006. (Cited on pages 25 and 76.)

[30] Wei-Neng Chen and Jun Zhang. Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *IEEE Transactions on Software Engineering*, 39(1):1–17, January 2013. (Cited on page 20.)

[31] Li-Te Cheng, Cleidson R.B. de Souza, Susanne Hupfer, John Patterson, and Steven Ross. Building collaboration into IDEs. *ACM Queue*, 1(9):40–50, 2003. (Cited on page 4.)

[32] Li-Te Cheng, Michael Desmond, and Margaret-Anne Storey. Presentations by programmers for programmers. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 788–792. IEEE Computer Society, 2007. (Cited on pages 100 and 101.)

[33] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989. (Cited on pages 100 and 120.)

[34] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 2011. (Cited on pages 109 and 118.)

[35] Jean M. Costa, Marcelo Cataldo, and Cleidson R. de Souza. The scale and evolution of coordination needs in large-scale distributed projects: implications for the future generation of collaborative tools. In *Proceedings of CHI 2011 (29th ACM Conference on Human Factors in Computing Systems)*, pages 3151–3160. ACM, 2011. (Cited on page 75.)

[36] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, Inc, 3d edition, 2008. (Cited on page 10.)

[37] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988. (Cited on page 15.)

[38] Isabella da Silva, Ping Chen, Christopher Van der Westhuizen, Roger Ripley, and André van der Hoek. Lighthouse: Coordination through emerging design. In *Proceedings of ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange*, pages 11–15. ACM Press, 2006. (Cited on page 94.)

[39] Laura Dabbish and Robert Kraut. Research note—Awareness Displays and Social Motivation for Coordinating Communication. *Information Systems Research*, 19:221–238, June 2008. (Cited on pages 74 and 75.)

[40] Barthélémy Dagenais and Harold Ossher. Mismar: A new approach to developer documentation. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 47–48. IEEE Press, 2007. (Cited on page 100.)

[41] Cleidson R. B. de Souza, Stephen Quirk, Erik Trainer, and David F. Redmiles. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proceedings of GROUP 2007 (International ACM SIGGROUP Conference on Supporting Group Work)*, pages 147–156. ACM, 2007. (Cited on page 94.)

[42] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a good software practice thwarts collaboration: the multiple roles of APIs in software development. In *Proceedings of FSE 2004 (12th ACM International Symposium on Foundations of Software Engineering)*, pages 221–230. ACM Press, 2004. (Cited on page 56.)

[43] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of GROUP 2003 (International ACM Conference on Supporting Group Work)*, pages 105–114. ACM Press, 2003. (Cited on page 20.)

[44] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference on Software Engineering)*, pages 241–250. ACM, 2008. (Cited on pages 21, 26, 27, 28, 29, and 75.)

[45] Cleidson R. B. de Souza and David F. Redmiles. The awareness network, to whom should i display my actions? and, whose actions should i monitor? *IEEE Transactions on Software Engineering*, 37(3):325–340, May 2011. (Cited on page 6.)

[46] Robert DeLine. Del.icio.us development tools. In *Proceedings of CHASE 2008 (International Workshop on Cooperative and Human Aspects of Software Engineering)*, pages 33–36. ACM, 2008. (Cited on page 138.)

[47] Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of VLHCC 2005 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pages 241–248. IEEE Computer Society, 2005. (Cited on page 139.)

[48] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of ECSCW 2007 (10th European Conference on Computer Supported Cooperative Work)*, pages 24–28. Springer, 2007. (Cited on page 94.)

[49] Joan DiMicco, David R. Millen, Werner Geyer, Casey Dugan, Beth Brownholtz, and Michael Muller. Motivations for social networking at work. In *Proceedings of CSCW 2008 (ACM conference on Computer Supported Cooperative Work)*, page 711, 2008. (Cited on page 75.)

[50] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of CSCW 1992 (ACM Conference on Computer-supported Cooperative Work)*, pages 107–114. ACM, 1992. (Cited on page 6.)

[51] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *CSCW*, 14(4):323–368, 2005. (Cited on page 40.)

[52] Jim Duggan, Jason Byrne, and Gerard J. Lyons. A task allocation optimizer for software construction. *IEEE Software*, 21(3):76–82, May 2004. (Cited on page 20.)

[53] Kevin Dullemond, Ben van Gameren, Margaret-Anne Storey, and Arie van Deursen. Fixing the 'out of sight out of mind' problem: One year of mood-based microblogging in a distributed software team. In *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)*, pages 267–276. IEEE Press, 2013. (Cited on page 148.)

[54] Eclipse Foundation. Mylyn. [Software]. Available: `https://www.eclipse.org/mylyn/` [Accessed: Jun 4, 2014], 2014. (Cited on pages 5 and 93.)

[55] Uwe Flick. *An introduction to qualitative research*. SAGE Publications, 5th edition, 2014. (Cited on pages 10 and 148.)

[56] Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006. (Cited on page 50.)

[57] Karl Fogel. *Producing Open Source Software*. O'Reilly Media, first edition, 2005. (Cited on pages 40, 43, 47, and 49.)

[58] Thomas Fritz. Determining Relevancy: How Software Developers Determine Relevant Information in Feeds. In *Proceedings of CHI 2011 (29th ACM Conference on Human Factors in Computing Systems)*, pages 1827–1830, 2011. (Cited on page 76.)

[59] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 175–184. ACM, 2010. (Cited on pages 16 and 56.)

[60] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of ESEC/FSE 2007 (6th ACM Joint Meeting on Foundations of Software Engineering)*, pages 341–350. ACM, 2007. (Cited on page 139.)

[61] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 385–394, 2010. (Cited on page 76.)

[62] Randall Frost. Jazz and the eclipse way of collaboration. *IEEE Software*, 24(6):114–117, 2007. (Cited on page 4.)

[63] Adrian Furnham. Response bias, social desirability and dissimulation. *Personality and Individual Differences*, 7(3):385 – 400, 1986. (Cited on pages 83 and 117.)

[64] Barney Glaser and Anselm Strauss. *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967. (Cited on pages 37 and 39.)

[65] Julia H. Grace, Deijn Zhao, and danah boyd. Microblogging: What and how can we learn from it? In *Proceedings of the CHI Workshop on Microblogging*. ACM, 2010. (Cited on page 139.)

[66] Irene Greif, editor. *Computer-supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. (Cited on page 4.)

[67] Rebecca Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996. (Cited on pages 4 and 20.)

[68] Alicia M. Grubb and Andrew Begel. On the perceived interdependence and information sharing inhibitions of enterprise software engineers. In *Proceedings of CSCW 2012 (ACM Conference on Computer Supported Cooperative Work)*, pages 1337–1346. ACM, 2012. (Cited on pages 29, 56, and 58.)

[69] Jonathan Grudin. Groupware and social dynamics: eight challenges for developers. *Commun. ACM*, 37:92–105, January 1994. (Cited on pages 62 and 69.)

[70] Carl Gutwin, Reagan Penner, and Kevin A. Schneider. Group awareness in distributed software development. In *Proceedings of CSCW 2004 (ACM Conference on Computer Supported Cooperative Work)*, pages 72–81, 2004. (Cited on pages 33, 41, and 47.)

[71] Anja Guzzi. The Bellevue design. `http://www.st.ewi.tudelft.nl/~guzzi/bellevue/bellevue-design.pdf`, September 2012. (Cited on page 86.)

[72] Anja Guzzi. Documenting and sharing knowledge about code. In *Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 1535–1538. IEEE Press, 2012. (Cited on page 11.)

[73] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in open source software development mailing lists. In *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)*, pages 277–286, 2013. (Cited on pages 11, 26, and 31.)

[74] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting developers' coordination in the IDE. In *Proceedings of CSCW 2015 (18th ACM Conference on Computer Supported Cooperative Work)*, page to be published, 2015. (Cited on pages 10, 11, 13, and 79.)

[75] Anja Guzzi and Andrew Begel. Facilitating communication between engineers with CARES. In *Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 1367–1370. IEEE Press, 2012. (Cited on pages 11 and 55.)

[76] Anja Guzzi, Andrew Begel, Jessica K. Miller, and Krishna Nareddy. Facilitating enterprise software developer communication with CARES. In *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)*, pages 527–536, 2012. (Cited on pages 11, 21, and 55.)

[77] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. Collective code bookmarks for program comprehension. In *Proceedings of ICPC*

2011 (19th IEEE International Conference on Program Comprehension), pages 101–110. IEEE Press, 2011. (Cited on pages 11 and 99.)

[78] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. Collective code bookmarks for program comprehension – online appendix. `http://www.st.ewi.tudelft.nl/~guzzi/pollicino/user-study-1/`, 2011. (Cited on pages 106, 108, and 109.)

[79] Anja Guzzi, Martin Pinzger, and Arie van Deursen. Combining micro-blogging and IDE interactions to support developers in their quests. In *Proceedings of ICSM 2010 (IEEE International Conference on Software Maintenance)*, pages 1–5, 2010. (Cited on pages 11 and 119.)

[80] Lile Hattori. *Change-centric Improvement of Team Collaboration*. PhD thesis, Università della Svizzera Italiana, February 2012. (Cited on page 4.)

[81] Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 235–238, 2010. (Cited on pages 4 and 94.)

[82] Lile Hattori, Michele Lanza, and Marco D'Ambros. A qualitative analysis of preemptive conflict detection. Technical Report 2011/05, University of Lugano, September 2011. (Cited on page 20.)

[83] Rajesh Hegde and Prasun Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 178–187. IEEE CS Press, 2008. (Cited on page 20.)

[84] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR cookbook: Mining a decade of research. In *Proceedings of MSR 2013 (10th Working Conference on Mining Software Repositories)*, pages 343–352. IEEE Press, 2013. (Cited on page 6.)

[85] Rebecca M. Henderson and Kim B. Clark. Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Administrative Science Quarterly*, 35(1):9–30, March 1990. (Cited on page 20.)

[86] James Herbsleb, Audris Mockus, Thomas Finholt, and Rebecca Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*, pages 319–328. ACM Press, 2000. (Cited on page 4.)

[87] James D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *Proceedings of FOSE 2007 (Future of Software Engineering)*, pages 188–198. IEEE Computer Society, 2007. (Cited on pages 4 and 5.)

[88] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. Collaboration in software engineering projects: A theory of coordination. In *Proceedings ICIS 2006 (International Conference on Information Systems)*, 2006. (Cited on pages 4 and 16.)

[89] James D. Herbsleb and Deependra Moitra. Global software development. *IEEE Software*, 18(2):16–20, 2001. (Cited on pages 4, 5, and 6.)

[90] Reid Holmes and Andrew Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)*, pages 23–26, 2008. (Cited on page 75.)

[91] Reid Holmes and Robert J. Walker. Customized awareness: recommending relevant external change events. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 465–474, 2010. (Cited on pages 6 and 76.)

[92] IBM. Rational Team Concert. [Software]. Available: `http://jazz.net/projects/rational-team-concert/` [Accessed: Jun 4, 2014], 2014. (Cited on pages 5 and 93.)

[93] Karen A Jehn and Priti Pradhan Shah. Interpersonal relationships and task performance: An examination of mediation processes in friendship and acquaintance groups. *Journal of Personality and Social Psychology*, 72(4):775, 1997. (Cited on page 6.)

[94] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of ESEC/FSE 2009 (7th ACM Joint Meeting on Foundations of Software Engineering)*, pages 111–120. ACM, 2009. (Cited on page 20.)

[95] Robert Johansen. *GroupWare: Computer Support for Business Teams*. The Free Press, 1988. (Cited on pages 5 and 6.)

[96] John and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 20(3):10:1–10:35, August 2011. (Cited on page 20.)

[97] Jeff Johnson, Teresa L. Roberts, William Verplank, David C. Smith, Charles H. Irby, Marian Beard, and Kevin Mackey. The Xerox Star: A retrospective. *IEEE Computer*, 22(9):11–26, 28–29, September 1989. (Cited on page 88.)

[98] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19(2):77–131, March 2007. (Cited on page 10.)

[99] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, pages 732–741. IEEE Press, 2013. (Cited on page 20.)

[100] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of ICSE 2011 (33rd ACM/IEEE International Conference on Software Engineering)*, pages 351–360, 2011. (Cited on page 33.)

[101] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of FSE 2006 (14th ACM International Symposium on Foundations of Software Engineering)*, pages 1–11. ACM, 2006. (Cited on pages 76, 94, and 139.)

[102] Laurie J. Kirsch. The Management of Complex Tasks in Organizations: Controlling the Systems Development Process. *Organization Science*, 7(1):1–21, January 1996. (Cited on page 20.)

[103] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 344–353. IEEE Computer Society, 2007. (Cited on pages 16, 26, 56, 59, 75, 138, and 139.)

[104] AndrewJ. Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015. (Cited on page 149.)

[105] Robert E. Kraut and Lynn A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, March 1995. (Cited on pages 4, 20, and 146.)

[106] Elmer C. Kubie. Recollections of the first software company. *IEEE Annals of the History of Computing*, 16:65–71, June 1994. (Cited on page 4.)

[107] Ko Kuwabara. A bazaar at the edge of chaos. *First Monday*, 5(3), 2000. (Cited on page 34.)

[108] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, 37(3):307–324, May 2011. (Cited on page 25.)

[109] Michele Lanza, Lile Hattori, and Anja Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Proceedings of CSMR 2010 (14th IEEE European Conference on Software Maintenance and Reengineering)*, pages 207–216. IEEE CS Press, 2010. (Cited on page 11.)

[110] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM/IEEE International Conference on Software Engineering)*, pages 492–501. ACM, 2006. (Cited on pages 4, 5, 25, 59, 75, and 100.)

[111] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineerng*, 10:311–341, 2005. (Cited on page 10.)

[112] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase*

Appeal, Make Better Design Decisions, and Teach through Design. Rockport Publishers, 2nd edition, January 2010. (Cited on pages 81 and 84.)

[113] Thomas R. Lindlof and Bryan C. Taylor. *Qualitative Communication Research Methods*. SAGE Publications, Inc., 2010. (Cited on page 16.)

[114] K. J. Lynch, J. M. Snyder, D. R. Vogel, and W. K. McHenry. The arizona analyst information system: Supporting collaborative research on international technological trends. In S. Gibbs and A. A. Verrijn-Stuart, editors, *Multi-User Interfaces and Applications: Proceedings of the IFIP WG 8.4 Conference*, pages 159–174. North-Holland, 1990. (Cited on page 5.)

[115] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994. (Cited on pages 4 and 23.)

[116] Bella Martin and Bruce Hanington. *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport Publishers, 2012. (Cited on pages 18 and 37.)

[117] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. (Cited on pages 5 and 148.)

[118] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of MSR 2009 (6th International Working Conference on Mining Software Repositories)*, pages 131–140. IEEE Computer Society, 2009. (Cited on page 20.)

[119] Michael C. Medlock, Dennis Wixon, Mark Terrano, Ramon L. Romero, and Bill Fulton. Using the RITE method to improve products: A definition and a case study. In *Proceedings of UPA 2002 (Usability Professionals Association)*, 2002. (Cited on page 83.)

[120] Jessica R Mesmer-Magnus and Leslie A DeChurch. Information sharing and team performance: a meta-analysis. *Journal of Applied Psychology*, 94(2):535, 2009. (Cited on page 6.)

[121] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of ICSE 2000 (21st ACM/IEEE International Conference on Software Engineering)*, pages 263–272, 2000. (Cited on pages 32, 33, 40, and 49.)

[122] Rolf Molich and Jakob Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, March 1990. (Cited on pages 80, 84, 85, and 147.)

[123] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the Eclipse IDE? *IEEE Software*, 23:76–83, 2006. (Cited on page 102.)

[124] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference on Software Engineering)*, pages 521–530. ACM, 2008. (Cited on page 4.)

[125] Kumiyo Nakakoji, Yunwen Ye, and Yasuhiro Yamamoto. Comparison of coordination communication and expertise communication in software development: Motives, characteristics, and needs. In *New Frontiers in Artificial Intelligence*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010. (Cited on pages 56 and 75.)

[126] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. In *Proceedings of WCRE 2010 (17th Working Conference on Reverse Engineering)*, pages 259 –268. IEEE CS Press, 2010. (Cited on page 33.)

[127] Jakob Nielsen. 10 usability heuristics for user interface design. `http://www.nngroup.com/articles/ten-usability-heuristics/` [Accessed: May 2014], January 1995. (Cited on pages 80, 81, 84, and 85.)

[128] Christopher Oezbek and Lutz Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *Proceedings of ICSM 2006 (23th IEEE International Conference on Software Maintenance)*, pages 64 –73. IEEE Press, 2007. (Cited on pages 100 and 101.)

[129] Michael Ogawa, Kwan-Liu Ma, Christian Bird, Premkumar T. Devanbu, and Alex Gourley. Visualizing social interaction in open source software projects. In *Proceedings of APVIS 2007 (6th International Asia-Pacific Symposium on Visualization)*, pages 25–32, 2007. (Cited on page 33.)

[130] Gary M. Olson and Judith S. Olson. Distance matters. *Human-Computer Interaction*, 15(2):139–178, 2000. (Cited on page 5.)

[131] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html` [Accessed: May 2014], 2005. (Cited on page 138.)

[132] Tim O'Reilly and Sarah Milstein. *The Twitter Book*. O'Reilly Media, Inc., 2009. (Cited on page 120.)

[133] Alex F. Osborn. *Applied Imagination: Principles and Procedures of Creative Problem-Solving,*. Creative Education Foundation, 1993. (Cited on page 15.)

[134] Michael J. Pacione, Marc Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering)*, pages 70–79. IEEE CS Press, 2004. (Cited on pages 109 and 118.)

[135] H. M. Parsons. What happened at Hawthorne? new evidence suggests the Hawthorne effect resulted from operant reinforcement contingencies. *Science*, 183(4128):922–932, March 1974. (Cited on page 83.)

[136] David Pattison, Christian Bird, and Premkumar Devanbu. Talk and work: a preliminary report. In *Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)*, pages 113–116. ACM, 2008. (Cited on page 33.)

[137] Dewayne E. Perry, Nancy Staudenmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994. (Cited on page 4.)

[138] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Leveraging crowd knowledge for software comprehension and development. In *Proceedings of CSMR 2013 (17th European Conference on Software Maintenance and Reengineering)*, pages 57–66. IEEE CS Press, 2013. (Cited on page 94.)

[139] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, pages 1295–1298. IEEE CS Press, 2013. (Cited on page 94.)

[140] Rafael Prikladnicki. Exploring propinquity in global software engineering. In *Proceedings of ICGSE 2009 (4th IEEE International Conference on Global Software Engineering)*, pages 133–142. IEEE Computer Society, 2009. (Cited on pages 5 and 148.)

[141] Tiago Proenca, Nilmax Moura, and André van der Hoek. On the use of emerging design as a basis for knowledge collaboration. *New Frontiers in Artificial Intelligence*, 6284:124–134, 2010. (Cited on page 20.)

[142] Michael K. Rabby and Joseph B. Walther. Computer-mediated communication effects in relationship formation and maintenence. In Daniel J. Canary and Marianne Dainton, editors, *Maintaining relationships through communication*, chapter 7, pages 141–162. Lawrence Erlbaum and Associates, 2003. (Cited on page 73.)

[143] Sarah Rastkar and Gail C. Murphy. Why did this code change? In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, pages 1193–1196. IEEE Press, 2013. (Cited on page 94.)

[144] Eric Raymond. *The Cathedral and the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999. (Cited on pages 32, 33, 34, 40, 43, and 48.)

[145] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of ICSE 2011 (33rd ACM/IEEE International Conference on Software Engineering)*, pages 541–550. ACM, 2011. (Cited on pages 32, 33, and 48.)

[146] Romain Robbes. On the evaluation of recommender systems with recorded interactions. In *Proceedings of SUITE 2009 (1st Workshop on Search-Driven*

Development-Users, Infrastructure, Tools and Evaluation), pages 45–48. IEEE Computer Society, 2009. (Cited on page 139.)

[147] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? the case of a smalltalk ecosystem. In *Proceedings of FSE 2012 (20th ACM International Symposium on the Foundations of Software Engineering)*, pages 56:1–56:11. ACM, 2012. (Cited on pages 21 and 26.)

[148] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012. (Cited on page 149.)

[149] Anita Sarma. A survey of collaborative tools in software development. Technical Report UCI-ISR-05-3, Institute for Software Research, University of California, Irvine, 2005. (Cited on page 93.)

[150] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering)*, pages 23–33. IEEE Computer Society, 2009. (Cited on pages 4, 25, 49, and 94.)

[151] Anita Sarma, David Redmiles, and André van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of FSE 2008 (16th ACM International Symposium on Foundations of Software Engineering*, pages 113–123. ACM Press, 2008. (Cited on pages 20 and 27.)

[152] Anita Sarma, David Redmiles, and André van der Hoek. Categorizing the spectrum of coordination technology. *IEEE Computer*, 43(6):61–67, June 2010. (Cited on pages 5, 20, 74, and 93.)

[153] Jeff Sauro. *A Practical Guide to the System Usability Scale: Background, Benchmarks and Best Practices*. CreateSpace, 2011. (Cited on page 93.)

[154] Jeff Sauro and James R. Lewis. When designing usability questionnaires, does it hurt to be positive? In *Proceedings of CHI 2011 (29th ACM Conference on Human Factors in Computing Systems)*, pages 2215–2224. ACM, 2011. (Cited on page 93.)

[155] Adrian Schröter, Jorge Aranda, Daniela Damian, and Irwin Kwan. To talk or not to talk: factors that influence communication around changesets. In *Proceedings of CSCW 2012 (ACM Conference on Computer Supported Cooperative Work)*, pages 1317–1326, 2012. (Cited on page 32.)

[156] Till Schümmer and Jörg M. Haake. Supporting distributed software development by modes of collaboration. In *Proceedings of ECSCW 2001 (7th European Conference on Computer Supported Cooperative Work)*, pages 79–98. Kluwer Academic Publishers, 2001. (Cited on pages 74 and 75.)

[157] Ken Schwaber. *Agile project management with Scrum*, volume 7. Microsoft press Redmond, 2004. (Cited on page 5.)

[158] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25:557–572, 1999. (Cited on page 32.)

[159] Emad Shihab, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. On the central role of mailing lists in open source projects: An exploratory study. In *New Frontiers in Artificial Intelligence*, volume 6284, pages 91–103. Springer Berlin Heidelberg, 2010. (Cited on pages 32 and 33.)

[160] Emad Shihab, Zhen Ming Jiang, and Ahmed E. Hassan. On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project. In *Proceedings of MSR 2009 (6th International Working Conference on Mining Software Repositories)*. IEEE, 2009. (Cited on page 139.)

[161] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008. (Cited on page 138.)

[162] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 23–34. ACM, 2006. (Cited on pages 16 and 56.)

[163] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software. In *Proceedings of IWPC 2005 (13th International Workshop on Program Comprehension)*, pages 173–175. IEEE Computer Society, 2005. (Cited on page 139.)

[164] Donna Spencer. Card sorting: a definitive guide. `http://boxesandarrows.com/card-sorting-a-definitive-guide/` [Accessed: May 2014], April 2004. (Cited on pages 14, 17, 36, and 37.)

[165] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. TODO or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference on Software Engineering)*, pages 251–260. ACM Press, 2008. (Cited on page 102.)

[166] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. How software developers use tagging to support reminding and refinding. *IEEE Transactions on Software Engineering*, 35:470–483, 2009. (Cited on pages 100 and 101.)

[167] Tasktop. Tasktop Dev. [Software]. Available: `http://www.tasktop.com/dev` [Accessed: Aug 1, 2014], 2014. (Cited on page 93.)

[168] Christoph Treude and Margaret-Anne Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *Proceedings of*

ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering). IEEE Computer Society, 2009. (Cited on page 138.)

[169] Christoph Treude, Margaret-Anne Storey, Kate Ehrlich, and Arie van Deursen. Web2SE: First workshop on web 2.0 for software engineering. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*. ACM, 2010. (Cited on page 138.)

[170] Mario Triola. *Elementary Statistics*. Addison-Wesley, 2006. (Cited on page 51.)

[171] Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. Adinda: A knowledgeable, browser-based IDE. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 203–206. ACM, 2010. (Cited on pages 11, 121, and 138.)

[172] Bart van Rompaey and Serge Demeyer. Estimation of test code changes using historical release data. In *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pages 269–278. IEEE Computer Society, 2008. (Cited on page 109.)

[173] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 1995. (Cited on page 100.)

[174] Alf Inge Wang. A process centred environment for cooperative software engineering. In *Proceedings of SEKE 2002 (14th International Conference on Software Engineering and Knowledge Engineering)*, pages 469–472. ACM, 2002. (Cited on page 4.)

[175] Robert S. Weiss. *Learning From Strangers: The Art and Method of Qualitative Interview Studies*. Free Press, 1995. (Cited on page 16.)

[176] Jim Whitehead. Collaboration in software engineering: A roadmap. In *Proceedings of FOSE 2007 (Future of Software Engineering)*, pages 214–225. IEEE Computer Society, 2007. (Cited on pages 5 and 138.)

[177] S. S. Yau, J. S. Colofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of COMPSAC 1978 (2nd IEEE CS International Computer Software and Applications Conference)*, pages 60–65. IEEE Computer Society Press, 1978. (Cited on page 21.)

[178] Andy Zaidman, Bart van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of ICST (1st International Conference on Software Testing, Verification, and Validation)*, pages 220–229. IEEE, 2008. (Cited on page 109.)

[179] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010. (Cited on page 33.)

Summary

Supporting Developers' Teamwork from within the IDE

Teamwork is fundamental in the production of software, because of the substantial complexity of software systems and the urge for fast time-to-market. Nevertheless, the difficulties in coordinating developers and teams and in making them collaborate are among the main reasons why the software industry has always been in crisis. Different researchers have repeatedly confirmed this problem, by conducting many studies in different contexts.

Achieving good teamwork is difficult for many reasons, for example, because communication in natural language is ambiguous, human memory cannot remember all of the project's details, and keeping track of what everyone is doing, even in a small group, is hard. These issues are exacerbated if developers are distributed across the globe, or even if they simply keep their office doors shut or work in different floors of a building.

Supporting teamwork is one of the most difficult-to-improve aspects of software engineering. Researchers have built theories on teamwork for software projects and have proposed a number of solutions to the practical challenges faced during collaborative software development. In particular, given that developers spend most of their time in the software environment in which they read, write, test, and design source code (also called integrated development environment—IDE), researchers have proposed a number of approaches to support teamwork in the IDE. Prominent examples (such as Jazz and Mylyn) are full-fledged platforms built on top of the IDE, which are aimed at transforming it into a comprehensive collaboration tool.

In this dissertation, we work in this context: developers' teamwork in the IDE. We consider the IDE because it also allows us to better connect with the final product of the development effort (*i.e.,* the source code). We work toward creating lightweight additions to the IDE, instead of heavyweight ones, because the latter may disrupt the development workflow, habits, and development process in place in a team, and they often require a steep learning curve. We address both collocated and distributed development settings, because what supports teamwork is not limited to one scenario.

Overall, the goal of our work is to devise lightweight IDE additions that can be seamlessly integrated in the development workflow of a variety of teams and that provide additional effective support to developer's teamwork, while requiring little learning time.

We structure our work in three parts: The first part investigates (through two studies) how developers currently experience teamwork, to pinpoint important pain points and understand the leeway for improvement; the second and third parts start from the results of the first one and explore two approaches each to deal with one of the key aspects of teamwork: information sharing.

Studies

How do developers experience collaboration in teamwork?

To better understand how to improve teamwork support in the IDE, we first investigated the current practice of industrial developers' working in teams. Although developers are able to find effective workarounds for situations considered as problematic in literature, they find it difficult and time-consuming to correctly understand the changes to source code made by other people; and they are especially frustrated when these changes are done by a project mate, since he could have informed them.

How is information shared in open source software projects?

In the second research question, we investigate the case of open source software development (OSS). We investigated the communication channel that is considered in literature as the hub of project communication in OSS projects: the development mailing list. We found that communication is currently scattered across several types of channels, such as issue repositories, code commits, and face-to-face talk. As a result, the mailing list seems to have lost its role of main communication channel in favor of the issue repository, which is closer to the code and offers more structured communication than noisy emails.

Reflection

By answering the first two research questions we learned that: (1) knowledge generally accepted in previous research should be verified, (2) information sharing is vital to support teamwork, (3) developers could retrieve part of the information they need, but it is not easily accessible, and (4) tools to support teamwork should be connected to source code.

Approaches

How can we expose existing information to support teamwork?

To expose information that is already available, but not easily accessible in the IDE, we created two IDE extensions and evaluated them with users. CARES is our first addition: It gives information about who one should contact to ask information about a specific piece of code. It shows contacts' information, their availability, and their photo; CARES proved to be effective and useful to developers. BELLEVUE, our second addition, improves the code change support offered by the IDE: Not only does it keeps visible what changed since the last update of the local code, it also seamlessly integrates code history in the editor and adds contact information similarly to CARES. Developers rated the design of BELLEVUE very positively.

How can we aid information creation to support teamwork?

Part of the information that would be useful to developers is neither recorded nor shared. For example, when they spend time understanding a piece of source code, developers do not usually leave any trace, so their precious insights are lost once the task is completed. We argued that this kind of information would be very valuable to support teamwork and we devised two IDE extensions for this task. Pollicino is a simple, lightweight solution based on sharing bookmarks in the code. Our experiment showed that it can be effective to generate information useful for other team members. James is an approach that lets developers share their comments in form of (Twitter-like) brief messages and automatically records developers interactions with the IDE. In an initial study with developers, we showed the feasibility of our approach and that developers are interested in having it.

Reflection

By answering the last two research questions we learnt that: (1) Lightweight IDE extensions, which require neither huge implementation effort nor long learning time, can support teamwork effectively, (2) showing developers' photos close to source code increases developers' sense of community, (3) popular IDEs have poor support for dealing with code changes, but this can be improved, and (4) tools can help team members creating additional information to support teamwork, without requiring major efforts by them.

Anja Guzzi

Riepilogo

Supportare il lavoro di squadra degli sviluppatori dall'interno dell'IDE

Il lavoro di squadra (*teamwork*) è fondamentale nella produzione di software a causa della notevole complessità dei sistemi software e dei tempi di commercializzazione molto brevi caratteristici dell'industria informatica. Tuttavia, le difficoltà nel coordinare gli sviluppatori e i loro gruppi di sviluppo (*team*) e nel farli collaborare sono tra i motivi principali per cui l'industria del software è sempre stata in crisi. Diversi ricercatori, conducendo vari studi in diversi contesti, hanno confermato ripetutamente questo problema.

Ottenere un buon lavoro di squadra è difficile per molti motivi. Per esempio, perché la comunicazione in linguaggio naturale è ambigua, perchè la memoria umana non può ricordare tutti i dettagli del progetto, e perchè tenere traccia di ciò che ognuno sta facendo è difficile, persino in un piccolo gruppo. Questi problemi sono aggravati se gli sviluppatori sono sparsi per il mondo o anche se, più semplicemente, tengono le porte dell'ufficio chiuse oppure lavorano in diversi piani dell'edificio.

Supportare il lavoro di squadra è uno degli aspetti più difficili da migliorare dell'ingegneria del software (*software engineering*). I ricercatori hanno formulato teorie sul lavoro di squadra nei progetti software e hanno proposto una serie di soluzioni alle difficoltà pratiche che si incontrano durante lo sviluppo collaborativo del software (*collaborative software development*). In particolare, dato che gli sviluppatori trascorrono la maggior parte del loro tempo in un ambiente software (chiamato anche l'ambiente di sviluppo integrato, o IDE) in cui leggono, scrivono, testano, e progettano il codice sorgente, i ricercatori hanno proposto una serie di approcci per sostenere il lavoro di squadra nell'IDE. Gli esempi più significativi (come Jazz e Mylyn) sono delle vere e proprie piattaforme costruite attorno all'IDE, le quali mirano a trasformare l'IDE in un ambiente collaborativo a tutto tondo.

In questa tesi, lavoriamo in questo contesto: il lavoro di squadra degli sviluppatori, all'interno dell'IDE (*within the IDE*). Consideriamo l'IDE perché permette anche di collegarsi meglio con il prodotto finale degli sforzi degli sviluppatori (ovvero il codice sorgente). Lavoriamo sulla la creazione di applicazioni aggiuntive e "leggere" (*lightweight*) per l'IDE, al posto di quelle "pesanti", perché queste ultime possono turbare il flusso di lavoro, le abitudini e il processo di sviluppo in atto in un team di sviluppatori e spesso richiedono una curva di apprendimento ripida. Rivolgiamo la nostra attenzione a gruppi di sviluppo sia collocati sia distribuiti, perché ciò che può sostenere il lavoro di squadra degli sviluppatori non è limitato ad un solo di questi scenari.

Riassumendo, l'obiettivo del nostro lavoro è quello di ideare applicazioni leggere da aggiungere all'IDE, che si possano integrare nel flusso di lavoro di diversi team di sviluppo e che forniscano un ulteriore supporto efficace al lavoro di squadra degli sviluppatori, richiedendo loro poco tempo di apprendimento.

Il nostro lavoro è strutturato in tre parti: la prima parte indaga (tramite due studi) come gli sviluppatori vivano attualmente il lavoro di squadra, con lo scopo di individuare i punti dolenti più importanti e di capire il margine di manovra di miglioramento; mentre la seconda e la terza parte iniziano dai risultati ottenuti dalla prima ed esplorano ognuna due strategie per affrontare uno degli aspetti fondamentali del lavoro di squadra: la condivisione delle informazioni.

Gli studi

Come viene attualmente vissuto il lavoro di squadra da parte degli sviluppatori?

Per capire meglio come migliorare il supporto al lavoro di squadra nell'IDE, abbiamo prima studiato le attuali prassi di lavoro degli sviluppatori industriali che lavorano in gruppi. Anche se gli sviluppatori sono in grado di trovare soluzioni efficaci per situazioni considerate problematiche dalla letteratura, trovano difficile e dispendioso capire correttamente le modifiche al codice sorgente fatte da altre persone; inoltre sono particolarmente frustrati quando queste modifiche sono fatte da un collega di progetto, dal momento che quest'ultimo avrebbe potuto informarli.

Come vengono condivise le informazioni in un progetto open source?

Con questa seconda domanda, studiamo le modalità dello sviluppo di software *open source* (OSS). Abbiamo studiato il canale di comunicazione che è considerato in letteratura come il fulcro della comunicazione in progetti OSS: la *mailing list* dedicata agli sviluppatori. Riscontriamo che attualmente la comunicazione si svolge tramite vari canali con tipologie distinte, come per esempio i commenti alle modifiche del codice sorgente (*code commit*) e il parlare faccia a faccia.

Osservazione

Rispondendo alle prime due domande di ricerca abbiamo imparato che: (1) lo stato delle cose derivato e generalmente accettato da precedenti ricerche va verificato, (2) la condivisione delle informazioni è vitale per sostenere il lavoro di squadra, (3) gli sviluppatori potrebbero recuperare parte delle informazioni di cui hanno bisogno, ma quest'ultime non sono facilmente accessibili, e (4) le applicazioni software che sostengono il lavoro di squadra dovrebbero essere collegate al codice sorgente.

Le strategie

Come esporre informazioni esistenti in modo da supportare il lavoro di squadra?

Per esporre informazioni che sono già disponibili, ma non facilmente accessibili nell'IDE, abbiamo creato due estensioni leggere dell'IDE che abbiamo poi valutato con degli utenti. CARES è la nostra prima estensione: fornisce informazioni su chi si dovrebbe contattare per chiedere informazioni su un determinato pezzo di codice. Mostra le informazioni dei contatti, la loro disponibilità e la loro foto; CARES si è dimostrato efficace ed utile per gli sviluppatori. BELLEVUE, la nostra seconda aggiunta, migliora il supporto per integrare le modifiche al codice offerto dall'IDE: non solo mantiene visibile ciò che è stato cambiato, rispetto al codice locale, dall'ultimo aggiornamento, ma integra anche perfettamente la cronologia del codice nell'editor e vi aggiunge delle informazioni di contatto analogamente a CARES. Gli sviluppatori hanno valutato il concetto di BELLEVUE molto positivamente.

Come creare nuove informazioni in modo da supportare il lavoro di squadra?

Parte delle informazioni che potrebbero essere utili per gli sviluppatori non sono né registrate né condivise. Ad esempio, durante il tempo impiegato per capire un pezzo di codice sorgente, gli sviluppatori di solito non lasciano alcuna traccia, così le loro preziose intuizioni vanno perse una volta che il lavoro è terminato. Sosteniamo che questo genere di informazione sarebbe molto prezioso per supportare il lavoro di squadra e abbiamo ideato due estensioni dell'IDE a questo scopo. POLLICINO è una soluzione semplice, leggera e basata sulla condivisione di segnalibri (*bookmark*) nel codice. Un nostro esperimento ha dimostrato che può essere efficace per generare informazioni utili per altri membri del team di sviluppo. JAMES è un approccio che consente agli sviluppatori di condividere le loro osservazioni in forma di brevi messaggi (simili a un *tweet*) e che registra automaticamente le loro interazioni con l'IDE. In una prima valutazione, abbiamo mostrato sia la fattibilità del nostro approccio sia il l'interesse degli sviluppatori ad averlo.

Osservazione

Rispondendo alle ultime due domande di ricerca abbiamo imparato che: (1) estensioni dell'IDE che sono leggere, che non richiedono un enorme sforzo di implementazione e neanche un lungo tempo di apprendimento, sono in grado di supportare il lavoro di squadra efficacemente, (2) mostrare le foto degli sviluppatori vicino al codice sorgente aumenta il senso di comunità tra gli sviluppatori, (3) le IDE più popolari hanno uno scarso supporto per gestire l'integrazione delle modifiche del codice, ma questo può essere migliorato, e (4) delle applicazioni software possono aiutare i membri di un team di sviluppo, creando ulteriori informazioni a supporto del lavoro di squadra, senza bisogno di sforzi particolari da parte loro.

Anja Guzzi

Samenvatting

Teamwerkondersteuning voor ontwikkelaars vanuit de IDE

Teamwerk is onmisbaar bij het produceren van software, vanwege de aanzienlijke complexiteit van softwaresystemen en de noodzaak tot snelle oplevering. De moeite echter die het kost om teams succesvol samen te laten werken en hun werkzaamheden te coördineren vormt één van de belangrijkste redenen dat de software-industrie zich nog altijd in een crisis bevindt. Verschillende onderzoekers hebben dit probleem herhaaldelijk bevestigd, middels uiteenlopende studies in diverse contexten.

Goed teamwerk is moeilijk om vele redenen, bijvoorbeeld omdat communicatie in natuurlijke taal ambigu is, het menselijk geheugen niet alle details van een project kan onthouden, en omdat bijhouden wat iedereen doet zelfs in een kleine groep ondoenlijk is. Deze problemen verergeren als ontwikkelaars verspreid over de wereld werken, of zelfs als ze enkel hun kantoordeuren dicht houden of op verschillende verdiepingen van een gebouw werken.

Het ondersteunen van teamwerk is één van de moeilijkste aspecten van software engineering. Onderzoekers hebben theorieën specifiek over teamwerk voor softwareprojecten opgesteld, en een aantal oplossingen voor de praktische uitdagingen tijdens collaboratieve softwareontwikkeling voorgesteld. Gegeven het feit dat ontwikkelaars het grootste deel van hun tijd doorbrengen in de software-omgeving waarin ze broncode lezen, schrijven, testen en ontwerpen (ook wel een geïntegreerde ontwikkelomgeving of IDE genoemd), hebben onderzoekers een aantal manieren voorgesteld om teamwerk te ondersteunen in de IDE. Prominente voorbeelden (zoals Jazz en Mylyn) zijn volwaardige platformen gebouwd boven op de IDE, met als doel die te transformeren tot alomvattend gereedschap voor samenwerking.

De context van dit proefschrift is het teamwerk van ontwikkelaars in de IDE. Wij kiezen de IDE, omdat dit ons in staat stelt om een beter verbinding te leggen met het eindproduct van de ontwikkeling (d.w.z. de broncode). We richten ons hierbij op lichtgewicht toevoegingen aan de IDE, in plaats van zwaardere, omdat dat laatste de manier van werken, de gewoontes en het ontwikkelproces van een team kunnen verstoren, en omdat ze vaak een steile leercurve hebben. We richten on zowel samenwerking op één locatie (*collocated*) als op gedistribueerde ontwikkeling, want wat teamwerk ondersteunt, is niet beperkt tot één scenario.

Het overkoepelende doel van ons werk is om lichtgewicht IDE-toevoegingen te ontwikkelen die naadloos geïntegreerd kunnen worden in de ontwikkelworkflow van diverse teams, die zorgen voor extra effectieve ondersteuning aan het teamwerk van een ontwikkelaar, terwijl ze weinig inwerktijd vereisen.

Wij delen ons werk op in drie delen: het eerste deel onderzoekt hoe ontwikkelaars momenteel teamwerk ervaren, om belangrijke pijnpunten op te sporen en te begrijpen waar ruimte voor verbetering is; het tweede en het derde deel beginnen bij de resultaten van het eerste deel en verkennen ieder twee strategieën om om te gaan met één van de belangrijkste aspecten van teamwerk: het delen van informatie.

Studies

Hoe ervaren ontwikkelaars samenwerking in teamwerk?

Om beter te begrijpen hoe ondersteuning voor teamwerk in de IDE kan worden verbeterd, hebben we eerst de huidige praktijk van industriële ontwikkelaars in teams onderzocht. Hoewel de ontwikkelaars in staat zijn om effectieve oplossingen te vinden voor situaties die in de literatuur als problematisch worden beschouwd, vinden ze het moeilijk en tijdrovend veranderingen te begrijpen die door andere mensen in de broncode zijn aangebracht; en ze zijn vooral gefrustreerd wanneer deze veranderingen zijn gedaan door een projectgenoot, omdat die hen had kunnen informeren.

Hoe wordt informatie in open source projecten gedeeld?

In de tweede onderzoeksvraag, onderzoeken we open source software ontwikkeling (OSS). We onderzochten het communicatiekanaal dat in de literatuur als de spil van de projectcommunicatie in OSS wordt beschouwd: de mailinglijst. We ontdekten dat de communicatie momenteel wordt verspreid over verschillende soorten kanalen, zoals de issue databases, veranderingen aan de code (*code commits*), en face-to-face gesprekken. Als gevolg hiervan, lijkt de mailinglijst haar rol als belangrijkste communicatiekanaal te hebben verloren ten gunste van de issue database, die dichter bij de code staat en meer gestructureerde communicatie biedt dan e-mails vermengd met ruis.

Reflectie

Door het beantwoorden van de eerste twee onderzoeksvragen hebben we geleerd dat: (1) kennis die in eerder onderzoek als algemeen geldend aanvaard werd, opnieuw moet worden gecontroleerd, (2) het delen van informatie essentieel is voor de ondersteuning van teamwerk, (3) ontwikkelaars een deel van de informatie die ze nodig hebben kunnen vinden, maar niet gemakkelijk, en (4) gereedschap ter ondersteuning van teamwerk een verbinding moet leggen met de broncode.

Strategieën

Hoe kunnen we bestaande informatie om teamwerk te ondersteunen blootleggen?

Om informatie bloot te leggen die al beschikbaar, maar niet gemakkelijk toegankelijk is in de IDE, creëerden we twee IDE-extensies en hebben we deze geëvalueerd met gebruikers. CARES is onze eerste extensie: Het geeft informatie over wie men moet contacteren om informatie te vragen over een specifiek stukje code. Het laat informatie over contacten zien, zoals hun beschikbaarheid en foto; CARES heeft bewezen effectief en nuttig voor ontwikkelaars te zijn. BELLEVUE, onze tweede extensie, verbetert de ondersteuning van de IDE voor veranderende code: niet alleen maakt het zichtbaar wat er veranderd is sinds de laatste update van de lokale code, maar het integreert ook naadloos code geschiedenis in de editor, en voegt contactgegevens toe op dezelfde wijze als CARES. Ontwikkelaars beoordeelden de ontwerpen van BELLEVUE zeer positief.

Hoe kunnen we het maken van informatie vergemakkelijken om teamwerk te ondersteunen?

Een gedeelte van de informatie die nuttig zou zijn voor ontwikkelaars is noch vastgelegd noch gedeeld. Wanneer ze bijvoorbeeld tijd doorbrengen met het doorgronden van een stukje broncode, laten ontwikkelaars meestal geen enkel spoor achter, zodat hun waardevolle inzichten verloren gaan als de taak is volbracht. We stellen dat dit soort informatie zeer waardevol is om teamwerk te ondersteunen, en daarom hebben we hiervoor twee IDE-extensies ontwikkeld. POLLICINO is een eenvoudige, lichtgewicht oplossing, gebaseerd op het delen van bladwijzers in code. Ons experiment toonde aan dat het effectief kan zijn om informatie te genereren die nuttig is voor de andere teamleden. JAMES is een aanpak die ontwikkelaars opmerkingen laat delen in de vorm van (Twitter-achtige) korte berichten, en die automatisch de interacties van ontwikkelaars met de IDE opneemt. Uit een eerste studie met ontwikkelaars is de haalbaarheid van onze aanpak gebleken, en ook dat ontwikkelaars geïnteresseerd zijn om deze extensie te gebruiken.

Reflectie

Door het beantwoorden van de laatste twee onderzoeksvragen hebben we geleerd dat: (1) lichtgewicht IDE-extensies, die noch een enorme ontwikkeldtijd, noch een lange inwerktijd nodig hebben, teamwerk effectief kunnen ondersteunen, (2) het tonen va foto's van ontwikkelaars in de buurt van de broncode het gemeenschapsgevoel verhoogt, (3) populaire IDE's slechte ondersteuning hebben voor het omgaan met wijzigingen in de code, maar dat dit verbeterd kan worden, en (4) tools teamleden kunnen helpen met het creëren van extra informatie ter ondersteuning van teamwerk, zonder grote inspanning van de teamleden zelf.

Anja Guzzi

Curriculum Vitae

June 25, 1985: Born

Faido (Switzerland)

Education

Nov 2009–Dec 2014: Ph.D in Software Engineering

Delft University of Technology (TU Delft), Delft, The Netherlands

Sep 2007–Jun 2009: M.Sc. in Informatics (summa cum laude)

University of Lugano (USI), Lugano, Switzerland.

Major in **Software Design**. Master thesis "*Supporting Collaboration Awareness in Multi-developer Projects*" under the supervision of Prof. dr. Michele Lanza.

Oct 2004–Jun 2007: B.Sc. in Informatics (magna cum laude)

University of Lugano (USI), Lugano, Switzerland

Work Experience

Jun 2012–Sep 2012: UX Researcher/Designer Intern

Microsoft, Redmond, United States

Jun 2011–Aug 2011: Research Intern

Microsoft Research, Redmond, United States

Dec 2007–Mar 2008: Software developer

AdulaNet, Bellinzona, Switzerland

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05