The Role of Social Media Artifacts in Collaborative Software Development

by

Christoph Treude
Dipl.-Wirt.Inform., University of Siegen, 2007

A Dissertation Submitted in Partial Fulfilment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

The Role of Social Media Artifacts in Collaborative Software Development

by

Christoph Treude
Dipl.-Wirt.Inform., University of Siegen, 2007

Supervisory Committee

Supervisory Committee

ABSTRACT

Social media mechanisms, such as wikis, blogs, tags and feeds, have transformed the way we communicate, work and play online. Many of these technologies have made their way into collaborative software engineering processes and modern software development platforms, either as an adjunct or integrated into a wide range of tools ranging from code editors and issue trackers to IDEs and web-based portals. Based on the results of several large scale empirical studies, this thesis presents findings on how social media artifacts, such as tags, feeds and dashboards, bridge lightweight and heavyweight task management in software development. Furthermore, this work shows how blogs, developer wikis and Q&A websites are changing the way software is documented. Based on these findings, the thesis describes a model that characterizes social media artifacts along several dimensions, such as content type, intended audience, and review mechanisms. The role of social media artifacts in collaborative software development lies in the timely dissemination of scenarios and concerns to a diverse audience through a process of implicit and informal collaboration, triggered by questions from users or articulation work. These findings lead to tool and process recommendations as well as the implementation of tools that leverage social media artifacts, and they indicate that tool support inspired by social media may play an important role in improving collaborative software development practices.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor Margaret-Anne (Peggy) Storey. Peggy's enthusiasm, passion and dedication have made this PhD extremely enjoyable. After meetings with Peggy, not only do I have a better idea of what my next step should be, but I always leave feeling excited to do it. Peggy has encouraged, motivated, and energized me throughout my PhD career, always managing to shed light on any challenge, making me see things in a different way. Her positive and optimistic attitude has encouraged and inspired me throughout these last five years.

I would also like to thank my co-supervisor Jens Weber. Jens was instrumental in introducing me to and then welcoming me to the University of Victoria. His suggestions and feedback have played an important part in this work.

I am very grateful to the members of my examining committee: Raymond Siemens' insights have provided me with a new interdisciplinary lens through which to view my work, and I have thoroughly enjoyed every single one of our meetings. Thank you also to Gail Murphy. I felt very privileged to have such an accomplished researcher as external examiner of my work. Her insightful and detailed comments were instrumental in forming the complete thesis.

Thank you to all the current and former members of the CHISEL research group at the University of Victoria. Being part of this supportive work environment has been an invaluable experience, and I have greatly enjoyed being surrounded by such brilliant and interesting people. Thank you to all my co-authors, both inside and outside CHISEL: Chris Parnin, Leif Singer, Brendan Cleary, Fernando Figueira Filho, Lars Grammel, Patrick Gorman, Martin Salois, Ohad Barzilay, Alexey Zagalsky, Gargi Bougie, Daniel German, Arie van Deursen, Li-Te Cheng, Jorge Aranda, Adrian Schröter, and Holger Schackmann. I have learned much from these collaborations, and I hope to continue many of them in the future. Also, a special thank you to the co-organizers of the Web2SE workshop series: Arie van Deursen, Andrew Begel, Kate Ehrlich, and Sue Black. Organizing two years of Web2SE has been a highly beneficial experience, and I am indebted to all co-organizers for guiding me through how to successfully run an international workshop.

Throughout my PhD, I have been fortunate to have the support from many experienced researchers who have inspired me, have provided me with opportunities to grow, and have given me invaluable advice. In particular, I would like to thank Martin Robillard, André van der Hoek, Arie van Deursen, Harold Ossher, Peri Tarr, Kelly Lyons, Greg Wilson, and Walid Maalej.

Thank you to Cassandra Petrachenko for the attention to detail in proof-reading countless papers as well as for all the administrative support, and to Omar Bahy Badreddin for proof-reading this entire document.

I would like to acknowledge the IBM Centers for Advanced Studies (CAS) not only for the financial and technical support, but also for providing me with access to some of IBM's greatest development teams. Thank you to Marcellus Mindel, Jennifer Collins, and Donna Hamilton for the administrative support, and to Jean-Michel Lemieux, Adrian Cho, Kevin McGuire, David Dewar, and Brian Wolfe for helping me formulate research questions, for allowing me collect results, and for always supporting this work. Also, thank you to all the developers that have participated in this work through interviews, surveys, and observations. I sincerely hope that my work has been beneficial to the development teams I have studied as well as the wider Software Engineering community.

Thank you to all the developers participating in the "Social Programmer Ecosystem" comprised of websites such as Stack Overflow, GitHub, and Coderwall. Your imagination and creativity are quietly revolutionizing how software is being developed, and you have opened up many new and exciting research areas.

This work would not have been possible without the love and support from friends and family. Thank you to my parents for their unwavering support in my academic endeavours, even buying my first computer even though they have never owned one themselves. Thank you to Pit Pietsch, Neil Chakrabarty, Thomas "Schnob" Maier, Thomas Fritz, Eric Finnis, and Andrew Slow for their friendship, for making me laugh, and for always being there when I needed them. Thank you to Alf and Joan Barrett for letting us adopt them as our "Canadian parents", and to Neil Barrett and Veronica Lefebvre for all their help and support.

Finally, thank you Nancy for all your support and understanding, for always believing in me, for being one of my toughest critics, for moving with me across the country when the research required it, and for always being there for me. None of this would have been possible without your love and support.

DEDICATION

To my dad (1939 – 2011)

Part I

Introduction and Literature Review

Chapter 1

Introduction

"The complexity of software is an essential property, not an accidental one."
– Frederick P. Brooks [25]

Software development is among the most complicated tasks performed by humans [25]. In a typical software development process, developers perform several different activities: they use numerous tools to develop software artifacts ranging from source code and models to documentation and test scenarios, they use other tools to manage and coordinate their development work, and they spend a lot of time communicating with other members on their team. Most tools used by software developers in their daily work are tailored towards individual developers and do not usually support team work. However, software is rarely developed by individuals and the success of software projects largely depends on the effectiveness of communication and coordination within teams [103]. Many software development teams struggle to address the challenges of collaborative development [127] in an environment of constantly changing requirements and a changing software development landscape. In particular, development teams lack informal communication channels [85] and tools that bridge technical and social aspects [45].

On the other hand, in recent years, social media has revolutionized how humans create and curate knowledge artifacts online [129]. For instance, Wikipedia[1], a free encyclopedia built collaboratively using wiki software, is an example where a large group of individuals come together to create and curate content on the web using social media technologies. Despite the lack of formal mechanisms to ensure the quality

[1]http://www.wikipedia.org/

and comprehensiveness of content, Wikipedia has now become the *de-facto* standard for encyclopedias [29]. Another example is the online photo management and sharing application, Flickr[2]. Without formal rules or processes, photos on Flickr are managed using the social media mechanism tagging [122]. Furthermore, social media, in particular the micro-blogging service Twitter[3], is starting to play a major role in day-to-day politics and events. For example, Twitter was instrumental during the 2011 uprising in Egypt [159] and in the aftermath of the 2011 earthquake and tsunami in Japan [1]. All of these examples show how large groups of individuals come together to create and curate content on the web effectively without formal rules and processes, using a variety of social media tools, such as wikis, tagging, and blogs.

This thesis investigates the extent to which social media mechanisms can address the challenges in collaborative software development. In recent years, several social media mechanisms have made their way into collaborative software engineering processes and modern software development platforms, either as an adjunct or integrated into a wide range of tools ranging from code editors and task management systems to integrated development environments (IDEs) and web-based portals. Based on the results of several empirical studies, this thesis presents how social media artifacts, such as tags, feeds, and dashboards, bridge lightweight and heavyweight task management in software development. Furthermore, this work shows how blogs, developer wikis, and Question and Answer (Q&A) websites are changing the way software is documented. These findings indicate that tool support inspired by social media may play an important role in improving collaborative software development practices.

1.1 Research Statement and Scope

The overarching goal of this research is to provide the various stakeholders involved in a software development project (e.g., software developers, team leads, managers, and customers) with better tool and process support.

In order to achieve this goal, we first have to understand how developers work together to produce software. Empirical software engineering – a rather young discipline – views software engineering as an empirical science, with the goal of better understanding the practice of software engineering [137]. Therefore, several large scale empirical studies with various professional software development teams have

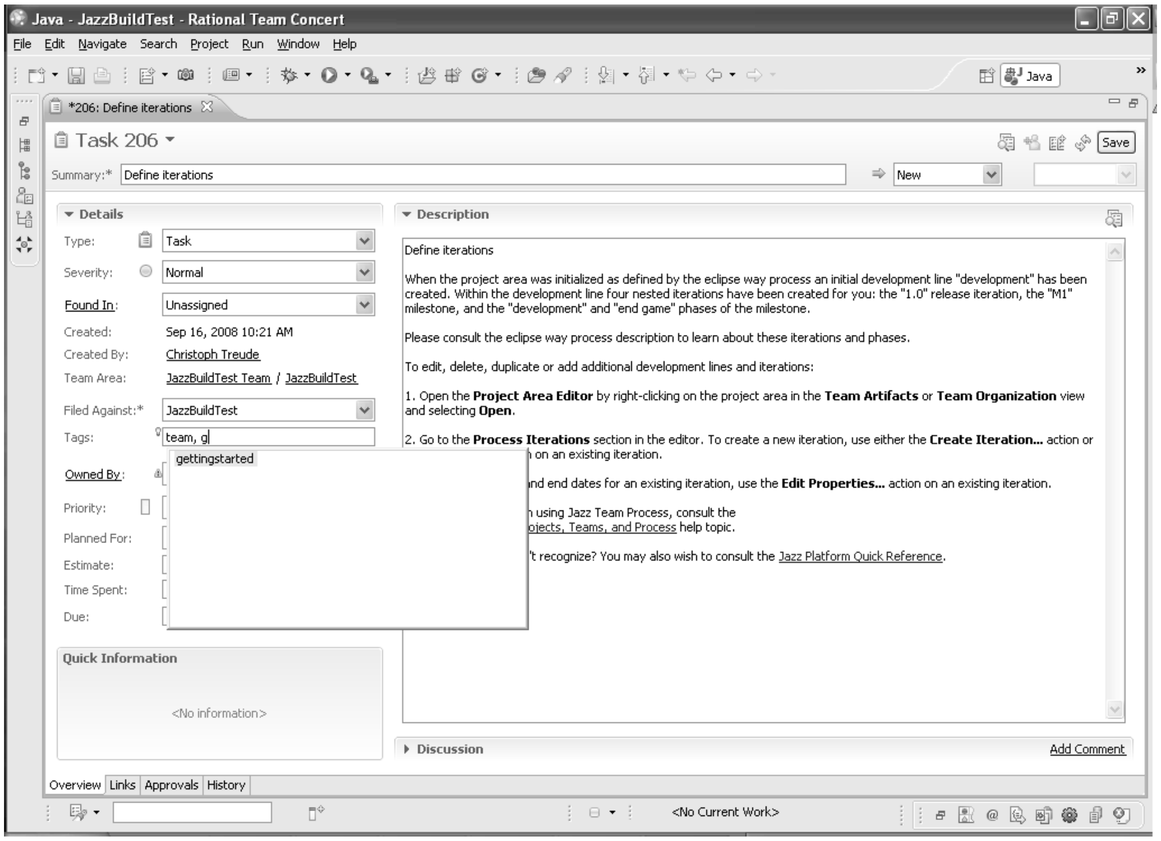
[2]http://www.flickr.com/

[3]https://twitter.com/

been conducted to gain first-hand insights into how professional software developers work, and what helps and hinders the success of collaborative development efforts.

The methods used in empirical software engineering research are inspired by social sciences as well as data mining, and they lead to the formulation of theories and frameworks that explain what the researchers observe and measure [53]. Based on these theories, we can affect evidence-driven change in software organizations that is grounded in scientific research. This research has followed a pragmatic approach (i.e., valuing practical knowledge over abstract knowledge) [123], and a mix of research methods – ranging from grounded theory and interviews to mining software repositories (MSR) – has been used to gain a better understanding of the complex nature of software development.

To scope this research, the analysis has been limited to two areas: task management and documentation.



Task 206

Summary:* Define iterations

New

Type: Task

Severity: Normal

Found In: Unassigned

Created: Sep 16, 2008 10:21 AM

Created By: Christoph Treude

Team Area: JazzBuildTest Team / JazzBuildTest

Filed Against:* JazzBuildTest

Tags: team, g

gettingstarted

Owned By:

Priority:

Planned For:

Estimate:

Time Spent:

Due:

Define iterations

When the project area was initialized as defined by the eclipse way process an initial development line "development" has been created. Within the development line four nested iterations have been created for you: the "1.0" release iteration, the "M1" milestone, and the "development" and "end game" phases of the milestone.

Please consult the eclipse way process description to learn about these iterations and phases.

To edit, delete, duplicate or add additional development lines and iterations:

1. Open the **Project Area Editor** by right-clicking on the project area in the **Team Artifacts** or **Team Organization** view and selecting **Open**.

2. Go to the **Process Iterations** section in the editor. To create a new iteration, use either the **Create Iteration...** action or
n on an existing iteration.

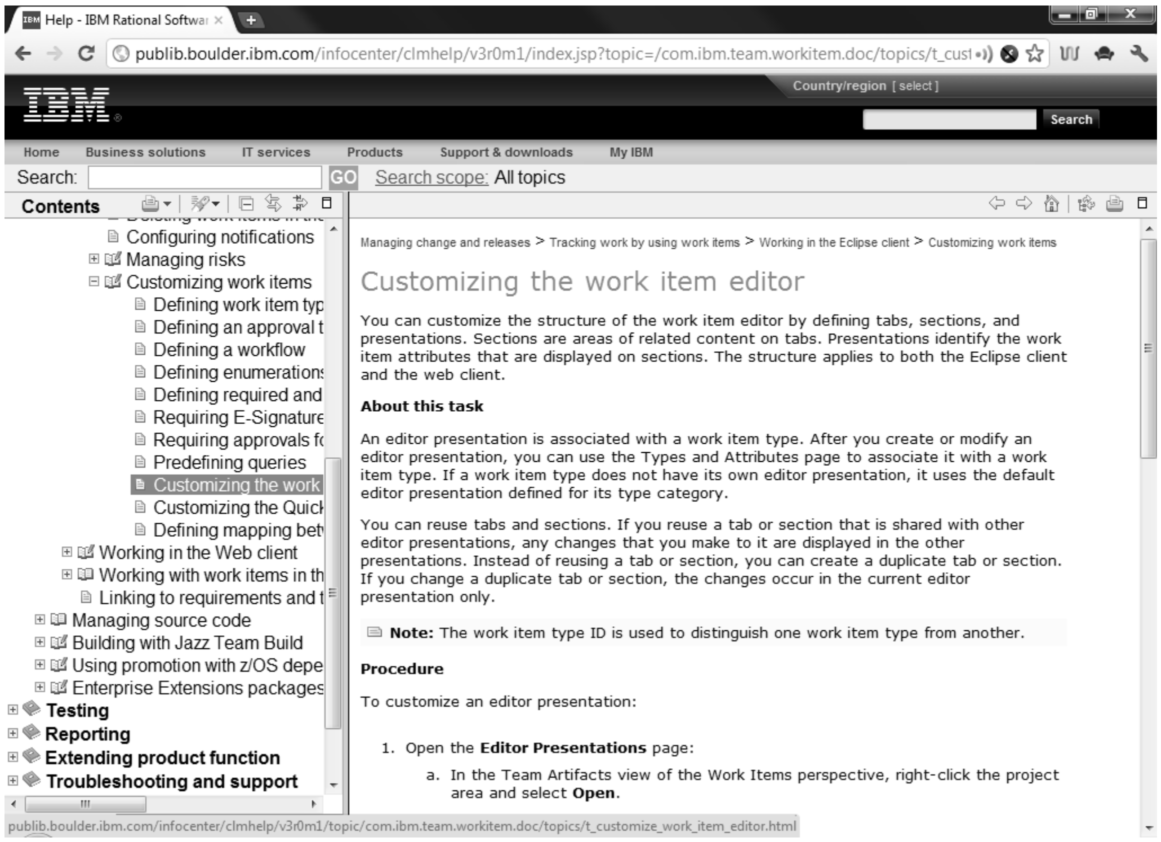
nd end dates for an existing iteration, use the **Edit Properties...** action on an existing iteration.

n using Jazz Team Process, consult the
ojects, Teams, and Process help topic.

't recognize? You may also wish to consult the Jazz Platform Quick Reference.

Figure 1.1: Work item interface in IBM's Jazz

Task Management. The management of software development tasks has long been


Help - IBM Rational Softwar
publib.boulder.ibm.com/infocenter/clmhelp/v3r0m1/index.jsp?topic=/com.ibm.team.workitem.doc/topics/t_cust
Country/region [select]
Search
Home
Business solutions
IT services
Products
Support & downloads
My IBM
Search:
GO
Search scope: All topics
Contents
Configuring notifications
Managing risks
Customizing work items
Defining work item typ
Defining an approval t
Defining a workflow
Defining enumeration
Defining required and
Requiring E-Signature
Requiring approvals fo
Predefining queries
Customizing the work
Customizing the Quick
Defining mapping bet
Working in the Web client
Working with work items in th
Linking to requirements and t
Managing source code
Building with Jazz Team Build
Using promotion with z/OS depe
Enterprise Extensions packages
Testing
Reporting
Extending product function
Troubleshooting and support
publib.boulder.ibm.com/infocenter/clmhelp/v3r0m1/topic/com.ibm.team.workitem.doc/topics/t_customize_work_item_editor.html

Managing change and releases > Tracking work by using work items > Working in the Eclipse client > Customizing work items

Customizing the work item editor

You can customize the structure of the work item editor by defining tabs, sections, and presentations. Sections are areas of related content on tabs. Presentations identify the work item attributes that are displayed on sections. The structure applies to both the Eclipse client and the web client.

About this task

An editor presentation is associated with a work item type. After you create or modify an editor presentation, you can use the Types and Attributes page to associate it with a work item type. If a work item type does not have its own editor presentation, it uses the default editor presentation defined for its type category.

You can reuse tabs and sections. If you reuse a tab or section that is shared with other editor presentations, any changes that you make to it are displayed in the other presentations. Instead of reusing a tab or section, you can create a duplicate tab or section. If you change a duplicate tab or section, the changes occur in the current editor presentation only.

Note: The work item type ID is used to distinguish one work item type from another.

Procedure

To customize an editor presentation:

1. Open the **Editor Presentations** page:
   a. In the Team Artifacts view of the Work Items perspective, right-click the project area and select **Open**.

Figure 1.2: Documentation example from IBM's Jazz

known to be a particularly challenging aspect of software development. Software development tasks are important cogs in the development process machine that need to be carefully aligned with one another, both in what they achieve and in their timing. Since tasks cross-cut technical and social aspects of the development process, the way they are managed has a significant impact on the success of a project.

As Frederick P. Brooks stated in what is now known as Brook's Law, *"Adding manpower to a late software project makes it later"* [26]. One problem is that industry and academia have not yet been able to create a task management system efficient enough to transform the addition of new team members into a faster and better software development process. Brook's Law has recently been confirmed by Meneely *et al.* [124], who also found that periods of accelerated team expansion are correlated with reduced software quality.

Software development environments typically have explicit tool support for managing tasks. For example, IBM's Jazz[4] [68] has tool support for managing "work items", where a work item is a generalized notion of a development task (see Figure 1.1 for an example). Work items are assigned to developers, are classified using pre-defined categories, and may be associated with other work items.

This thesis investigates the role of social media artifacts, such as tags, dashboards, and feeds, in task management for software developers. To that end, data from software development teams using IBM's Jazz as well as from projects using the Google Code Issue Tracker[5] has been collected and analyzed. IBM's Jazz is an extensible technology platform that helps teams integrate tasks across the software lifecycle. The software development team collaboration tool built on top of Jazz is called Rational Team Concert (RTC). As Jazz is one of the first environments to tightly integrate social media artifacts, such as tags, dashboards, and feeds, into the IDE, it enables studying the role of social media artifacts in software development.

Documentation. Similar to task management, documentation has been a challenge in software development for a long time. As David Parnas wrote, *"Poor documentation is the cause of many errors and reduces efficiency in every phase of a software product's development and use."* [134].

There have been many efforts to promote the creation and maintenance of good documentation, however, all too often, documentation is absent or incomplete. When documentation is written, it quickly becomes stale. This stagnation is often the root of mistrust, which can lead to documentation being rarely consulted in practice [109]. Further, documentation is often spartan [143], leaving developers with insufficient examples or explanations.

For customers, documentation is typically made available through help menus in their software (see Figure 1.2 for an example). This documentation focuses on describing each function of a software product in detail, but often falls short in explaining why and how a certain functionality should be used.

This thesis examines the role that social media artifacts, such as wikis, blogs, and Question and Answer (Q&A) threads, can play in software documentation.

[4]https://jazz.net/
[5]http://code.google.com/

Data from IBM's Jazz project as well as from the documentation of several open source projects has been collected and analyzed.

For both task management and documentation, I have investigated the role of social media artifacts in different contexts. For both areas, I have studied software development teams using IBM's Jazz. In addition, to validate the results with teams that are not using IBM's Jazz, the findings on task management have been confirmed with teams using the Google Code Issue Tracker, and the findings on documentation have been confirmed through studies of blogs and Q&A websites. While IBM's Jazz is closed source (i.e., the source code is not shared with the public), Google Code hosts open source projects (i.e., projects for which the source code is available to everyone). Blogs and Q&A websites span closed source and open source.

I define *social media artifact* in the context of software development as follows:

> Social media artifacts are tangible artifacts created through social media tools as part of informal individual or collaborative processes in software development.

In software development, the main difference between social media artifacts and traditional artifacts is that the former can be freely configured by everybody participating in the development, whereas the latter can only be configured by a "gatekeeper" – a role that is typically fulfilled by the project administrator or development manager. The social media tools used to create social media artifacts are characterized by an underlying "architecture of participation" that supports crowd-sourcing as well as a many-to-many broadcast mechanism [129]. The design of social media artifacts and tools supports and promotes collaboration, often as a side effect of individual activities, and furthermore democratizes who participates in activities. Social media tools and artifacts in software development are inspired by the success of social media, defined as a group of Internet-based applications, built on the ideological and technological foundations of information sharing, interoperability, user-centered design, and collaboration on the Internet [99].

1.2 Research Questions

The research presented in this thesis is motivated by the following three questions.

- What role do social media artifacts play in collaborative software development?

- How can tools for software developers leverage the knowledge from social media artifacts?
- How do social media artifacts interplay with other development artifacts?

I investigated these research questions for task management as well as for documentation.

1.3 Contributions

This work makes four overarching contributions:

Several large scale empirical studies of software development teams. Empirical software engineering aims to understand the practice of software engineering by treating software engineering as an empirical science [137]. As part of this thesis, I have conducted several large scale empirical studies with several professional software development teams in order to understand the role of particular social media artifacts in these teams.

A model of social media artifacts in collaborative software development. Based on the empirical studies, I developed a model of social media artifacts in collaborative software development. This model aims at explaining the role of social media artifacts by pointing out several dimensions along which social media artifacts differ from traditional software development artifacts. The model shows that the role of social media artifacts in collaborative software development lies in the timely dissemination of scenarios and concerns to a diverse audience through a process of implicit and informal collaboration, triggered by questions from users or articulation work.

Tool and process recommendations. Based on the empirical studies, I have also made several tool and process recommendations to the teams under study. These recommendations are discussed as part of reporting the results from the empirical studies in the following chapters.

New tools for developers that leverage social media artifacts. As part of this thesis work, I have developed two tools that leverage the information stored in social media artifacts. These tools – ConcernLines and WorkItemExplorer

– aim at helping developers understand their use of social media artifacts by making data exploration flexible and interactive.

1.4 Thesis Outline

This thesis is structured in four parts (see also Figure 1.3). In the following, the content of each part is highlighted, and it is shown how each part relates to papers that were published as part of the thesis work.

Part I: Introduction and Literature Review

Content. After introducing the topic, a review on background and related work is given (Chapter 2). Research related to this thesis can be divided into work on the importance of social aspects in collaborative software development (Section 2.1), and research on the use of social media by software developers (Section 2.2). Mostly because social media has only recently started to attract the attention of professional software developers, research on the role of social media in software development is limited and usually focuses on one particular aspect or tool. This research takes a more comprehensive view by studying the role of social media artifacts through various large scale empirical studies, looking at software development teams in different contexts using different kinds of social media in their work. This work also focuses on the interplay of traditional software development mechanisms, such as formal task management and help documentation, with social media mechanisms.

Publications. The main theme of this work has previously been published as a Doctoral Symposium paper at the International Conference on Software Engineering (ICSE) 2010 [168]. Parts of Section 2.1 have been published as a technical report [183], and Section 2.2 builds on publications related to the International Workshop on Web 2.0 for Software Engineering (Web2SE) [179, 180, 181, 182] as well as a position paper at the workshop on the Future of Software Engineering Research (FoSER) 2010 [163].

Part II: Task Management

Content. The second part of this thesis presents the findings on the role of social media artifacts in task management. I have explored how the social computing mechanism, tagging, is used to communicate matters of concern in the management of development tasks. In two longitudinal studies (over 36 and 12 months respectively) with IBM's Jazz development team and several other large development teams that use Jazz, I showed that the tagging mechanism was eagerly adopted, and that it has become a significant part of many informal processes, such as planning and awareness (Section 3.1). This work has led to the development of a tool that aims to surface the use of tags over time and to answer questions such as "Which components played a key role during the last beta release?" (ConcernLines, Section 3.2).

In addition, I have explored how extensive awareness is accomplished through a combination of highly configurable project, team, and contributor dashboards as well as individual event feeds. The results presented in this thesis stem from an empirical study of several large development teams, with a detailed study of IBM's Jazz team and additional data from another four teams. Section 4.1 presents how dashboards become pivotal to task prioritization in critical project phases and how they stir competition while feeds are used for short term planning. These findings indicate that the distinction between high-level and low-level awareness is often unclear and that integrated tooling could improve development practices. To address this gap, WorkItemExplorer, an interactive exploration environment for the visualization of software development tasks, was developed (Section 4.2).

Chapter 5 presents the findings pertinent to software developers using Google Code. Based on the analysis of 1,000 projects using the Google Code Issue Tracker, the findings on the role of tags in task management were confirmed, and the work suggests that social media mechanisms, such as tags, dashboards, and feeds, can play a major role in improving collaborative software development processes.

Publications. The findings on tagging (Section 3.1) have been published in Transaction on Software Engineering (TSE) [178] as an extension of a paper at ICSE 2009 [173]. The work has also been replicated by a group of researchers from Italy [31]. Short papers at ICSE 2010 [175] and at Web2SE 2010 [176] further describe details of software developers' use of tags. The findings on dashboards and feeds (Section 4.1) have been published at ICSE 2010 [174]. ConcernLines and WorkItemExplorer have been published as tool demo papers at ICSE 2009 [172] and ICSE 2012 [171], respectively. The work on WorkItemExplorer was done in collaboration with Lars Grammel and Patrick Gorman at the University of Victoria in Canada. At the time of writing, the findings related to the Google Code Issue Tracker have not been published yet.

Part III: Documentation

Content. The third part of this thesis presents the findings on the role of social media artifacts in software documentation. With the rise and wide accessibility of social media sites, such as blogs, Q&A websites, and developer forums, a new culture and philosophy has emerged that is changing the way we search for documentation, where we find it, and how we write it for others. Developers can now create and communicate knowledge and experiences without relying on a central authority to provide official documentation. Any content created by a developer is just a web search away. This phenomenon can be described as crowd documentation – documentation that is written by many and read by many. As a result, software companies are faced with a plethora of media forms, such as wikis, blogs, and Q&A websites, to disseminate knowledge to their customers. There is little advice on how these different media forms can be used effectively, and what knowledge is best represented through which channel. Using grounded theory [37], I have studied the documentation practices of IBM's Jazz team and I have developed a model that characterizes documentation artifacts along several dimensions, such as content type, feedback options, and review mechanisms (Chapter 6).

To understand whether crowd documentation via blogs, wikis, and Q&A websites can replace or augment more traditional forms of documentation, we have conducted research on the prevalence, nature, and impact of the documentation created by the crowd. In a study of the jQuery API, we found that close to 90% of the API methods are documented on software development blogs (Section 7.1), and in a qualitative analysis of content on Stack Overflow[6], a popular Q&A website for programmers, we found that the site is particularly effective at providing code reviews and answers to conceptual questions (Section 8.1). These findings indicate that social media is more than a niche in documentation, that it can provide high levels of coverage (Section 8.2), and that it gives readers a chance to engage with authors.

Publications. Chapter 6 has been published at the Symposium on the Foundations of Software Engineering (FSE) 2011 [177]. A position paper published at the workshop on the Future of Collaborative Software Development (FutureCSD) 2012 [170] outlines the motivation for the studies described in Chapters 7 and 8, and the work has been published at Web2SE 2011 [136] (Section 7.1) and ICSE 2011 [169] (Section 8.1).

[6]http://stackoverflow.com

Section 8.2 is under submission at the time of writing. The work in Chapters 7 and 8 was done in collaboration with Chris Parnin at the Georgia Institute of Technology in Atlanta, Georgia, United States, as well as with Ohad Barzilay at Tel Aviv University in Tel Aviv, Israel.

Part IV: Implications

Content. The last part of this thesis presents the implications of this work. First, in Chapter 9, a model that characterizes social media artifacts and contrasts them with formal tools along several dimensions, such as content type, intended audience, and time sensitivity, is presented. This model emerged out of the grounded theory study described in Chapter 6. The model is later refined based on the findings from the other studies described in this thesis. I conclude that the role of social media artifacts in collaborative software development is the timely dissemination of scenarios and concerns to a diverse audience through a process of implicit and informal collaboration, triggered by questions from users or articulation work (Chapter 10).

Publications. A preliminary version of the model presented in Chapter 9 was published at FSE 2011 [177].

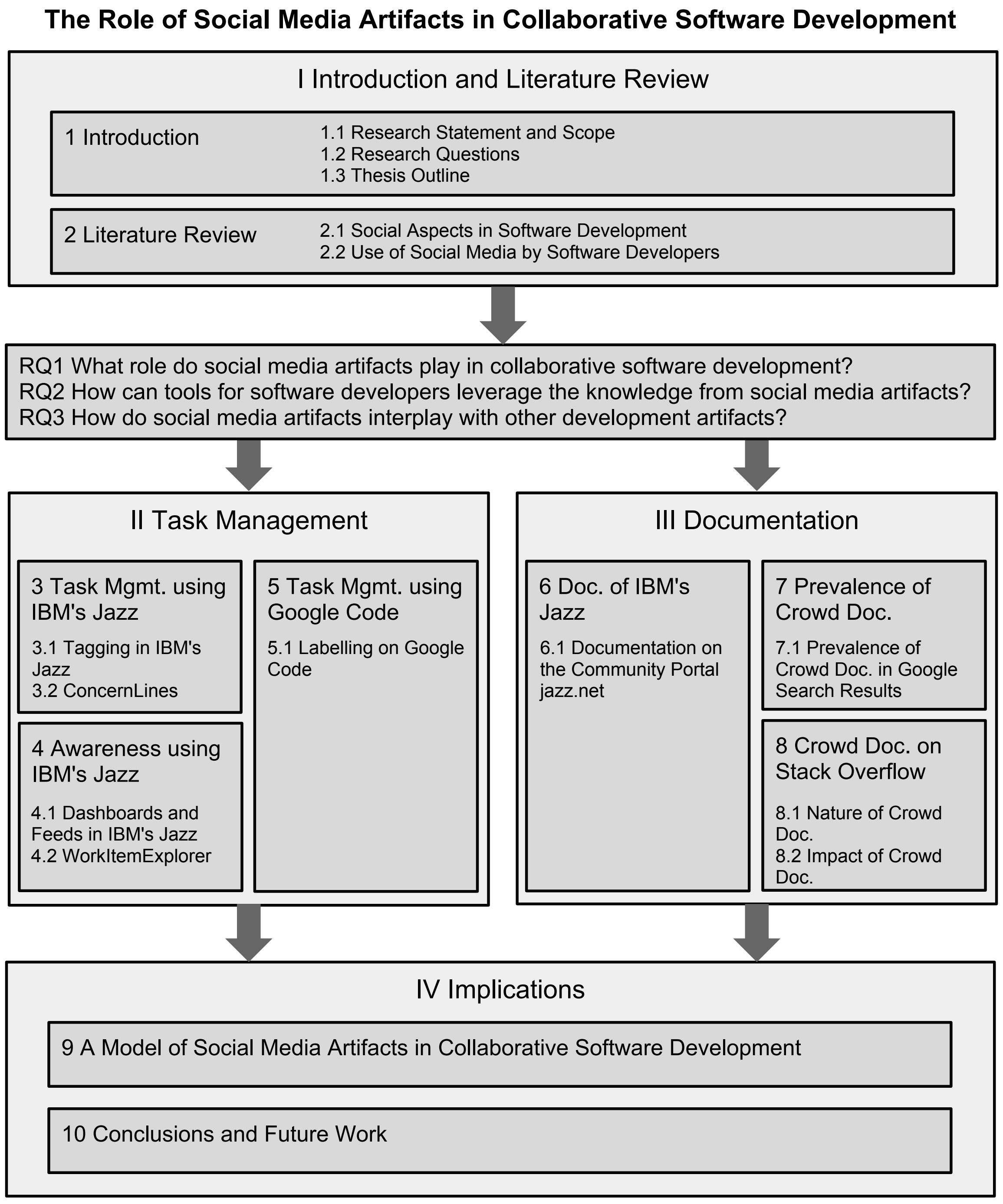

The Role of Social Media Artifacts in Collaborative Software Development
I Introduction and Literature Review
1 Introduction
1.1 Research Statement and Scope
1.2 Research Questions
1.3 Thesis Outline
2 Literature Review
2.1 Social Aspects in Software Development
2.2 Use of Social Media by Software Developers
RQ1 What role do social media artifacts play in collaborative software development?
RQ2 How can tools for software developers leverage the knowledge from social media artifacts?
RQ3 How do social media artifacts interplay with other development artifacts?
II Task Management
3 Task Mgmt. using IBM's Jazz
3.1 Tagging in IBM's Jazz
3.2 ConcernLines
4 Awareness using IBM's Jazz
4.1 Dashboards and Feeds in IBM's Jazz
4.2 WorkItemExplorer
5 Task Mgmt. using Google Code
5.1 Labelling on Google Code
III Documentation
6 Doc. of IBM's Jazz
6.1 Documentation on the Community Portal jazz.net
7 Prevalence of Crowd Doc.
7.1 Prevalence of Crowd Doc. in Google Search Results
8 Crowd Doc. on Stack Overflow
8.1 Nature of Crowd Doc.
8.2 Impact of Crowd Doc.
IV Implications
9 A Model of Social Media Artifacts in Collaborative Software Development
10 Conclusions and Future Work

Figure 1.3: Thesis outline

Chapter 2

Literature Review

Work related to this research can be divided into work on social aspects in software development (Section 2.1), and work on the use of social media by software developers (Section 2.2). In this chapter, both areas are reviewed in detail. Related work on task management and documentation is presented as part of the corresponding chapters in Parts II and III of this thesis.

2.1 Social Aspects in Software Development

There are several strands of research that have considered the impact of social aspects in software development. Researchers in many areas recognize that software development processes are more than writing source code, and that "articulation work" [125] must be supported in a software engineering project. According to Gerson and Star [72]: *"Articulation consists of all tasks needed to coordinate a particular task, including scheduling sub-tasks, recovering from errors, and assembling resources."* Other examples of articulation work include discussions about design decisions, assigning bug fixing tasks to developers, and deciding on interfaces.

In the following, I first review empirical studies on social aspects in software development (Section 2.1.1) before discussing related work on the importance of articulation work (Section 2.1.2) and work on the need for informal tool support for software developers (Section 2.1.3). Related work was identified through targeted web searches as well as searches of pertinent digital libraries, and by following the references in the papers found.

2.1.1 Empirical Studies on Social Aspects in Software Development

The trend to globalization has influenced the way software is designed, constructed, and tested. Herbsleb and Moitra [87] point at several problem dimensions that arise in global software development, such as strategic and cultural issues, which require additional work in a software development process. The results of a study conducted by Herbsleb *et al.* [86] indicate that software development takes longer when development teams are not co-located. Also, more developers are required to achieve the same results. The authors analyzed survey data and software artifacts, and they conclude that remote colleagues were not perceived to be helping when the workload got heavy, i.e., when articulation work was most important.

Particular coordination breakdowns are revealed by Catalado *et al.* [32]. The authors observed four different cases of breakdowns that occurred despite the presence of procedures and rules to support coordination. The breakdowns could be contributed to a lack of communication, unclear dependencies, circular dependencies, and schedule changes. Overall, they found that the importance of documentation in distributed software development was not clear to all developers and that important changes were not necessarily explained to other team members.

Globalization of software development also leads to cultural differences between the developers which can impede work. Some difficulties caused by cultural differences were pointed out in a study by Halverson *et al.* [80]. Using data gained from interviews and the inspection of task management systems at IBM, they found a list of social issues: conflicting views on whether a bug was really a bug, avoiding breaking another developer's code unnecessarily, figuring out what had caused broken code and who to talk to about it, and treating something as a technical problem that was really a social or cultural problem.

Since a lot of the knowledge required for software development exists only in the heads of developers rather than on paper or in files, all processes related to knowledge management play a central role in software development. The term knowledge management refers to the activities used to identify, create, represent, and distribute knowledge. LaToza *et al.* [108] refer to the implicit knowledge in software development as mental models of the software. In software development processes, expertise must be managed and coordinated in order to leverage its potential. A study done by Faraj and Sproull [59] investigated 69 software development teams with regard to their approach to knowledge management. The main result of this study is evidence for a strong relationship between coordination and team performance.

The importance of source code in software development with regard to social aspects is highlighted in a paper by de Souza *et al.* [45]. They claim that software source code is both a social and a technical artifact and that dependencies do not only exist between artifacts but also between developers. A study of software artifacts using a visualization tool did in fact show that dependencies between artifacts and between developers were intertwined.

The results of a study by Chudge and Fulton [34] point to *"predominantly social, rather than technical, problems"* in relationships between clients and developers on software development projects. Their study looked at requirements change practice in the British industry, putting an emphasis on safety-related software development. They conducted two case studies with industrial partners. As a result, they state that problems in the professional relationships between client and developer are mainly caused by social aspects.

2.1.2 Importance of Articulation Work

Articulation work cannot be narrowed down to certain activities during the software development process but rather affects all of them. Therefore, research has been conducted in order to determine how articulation work supports or does not support certain activities.

An activity that requires a lot of articulation work is task allocation, i.e., the assignment of a given task to a certain developer. Process models in this area do not take interpersonal interactions and social aspects into account. However, the problem of task allocation is closely related to the network structures of developers in a company. A study done by Amrit [10] gives supporting evidence for that. He measured network density, centralization, and team performance in his study and concludes that the performance of teams is positively related to the density of an interpersonal network.

Another situation in which articulation work is of particular importance is failure of plans. Rönkkö *et al.* [148] did a field study of a distributed software development project that showed that planning is an integral part of software development, and that articulation work is especially needed when plans do not work out. Related to that, a paper by Bendifallah and Scacchi [17] focused on articulation work in software maintenance. The authors found that maintenance is an activity that cannot be planned, and therefore requires a lot of articulation work.

Unless tools specifically created for articulation work are used, configuration management tools are often in place to coordinate the software development process. However, these tools have significant shortcomings when stretched to do more than what they were intended for: Grinter [75] lists the lack of representation of the work itself, insufficient support for individual developers and teams, and inappropriate built-in assumptions about the workflow as major challenges. Therefore, a lot of informal practices are applied to support articulation work.

2.1.3 Need for Informal Tool Support

Several studies point to the need for informal tool support for software developers. In their paper titled "Splitting the Organization and Integrating the Code", Herbsleb and Grinter [85] report on a case study on what they identified as the most difficult part of geographically distributed software projects: integration. They conducted ten interviews with managers and technical leads to gather information about perceived challenges of distributed development, followed by a second round of eight interviews that focused on integration explicitly. Their results show that coordination problems were greatly exaggerated across sites, largely because of the breakdown of informal communication channels. They conclude that distributed development may imply the necessity of stable plans, processes, and specifications. On the other hand, the inherently unpredictable aspects of projects require communication channels that can be invoked spontaneously.

Several tools for communication in collaborative software development have been proposed. An early introduction of instant messaging into the software development workplace is described by Herbsleb *et al.* [84]. They introduced the tool and gathered usage data via automatic logging on the server, which included logins, logouts, joining and leaving groups, as well as group chat messages. About two dozen semi-structured interviews were conducted with users, and two small focus group sessions were held to get feedback. The evaluation showed that the combination of features had some potential to help distributed teams overcome the lack of context and absence of informal communication, two of the problems that make distributed work difficult. However, there were adoption problems, in particular the perception that chatting is not real work.

Making communication channels public generates a gap between private and public work in collaborative software development, as observed in a study by de Souza *et al.* [46]. They conducted an ethnographic study for eight weeks, making observations and collecting information about several aspects of a software development team. Additional data was collected from manuals and process descriptions as well as training documentation for new developers and problem reports. They conclude that the transition from private to public work needs to be handled carefully.

A study by Lindqvist *et al.* [114] looked primarily at communication in global software development. The authors conducted interviews at Ericsson, asking questions about product structure, organizational structure, communication, and different ways of working. They found that a lack of continuity in communication and of informal communication made it hard to identify important issues at remote sites. This led to an underestimation of problems at remote sites. Co-workers unaccustomed with distributed development often used mail for communication with remote sites. However, the use of asynchronous tools, such as mail, created delays in communication.

Complementary to that, Woit and Bell [191] found that developers believe themselves to be significantly less effective in a distributed environment because of lack of traditional non-verbal cues. The authors conducted a survey to explore effectiveness of non face-to-face communication with students engaged in distance learning courses that required them to work together to complete software development tasks.

The lack of informal tool support was also observed in requiremens engineering. Damian and Zowghi [44] conducted a study on the impact of stakeholders' geographical distribution on requirements engineering in a multi-site organization. They collected data through the inspection of documents, observations of meetings, and semi-structured interviews. The authors identified several challenges in requirements engineering that can be attributed to the geographical distribution of stakeholders, such as diversity in customer culture and business, achieving appropriate participation of users, and lack of informal communication.

2.1.4 Summary

A recurring theme in studies on social aspects in software development is the need for informal tool support. Even if some informal tool support is present, there is often a lack of ways to bridge formal and informal tool support, and developers need to appropriate existing tools to fit their needs.

Social media mechanisms, such as wikis, tags, and blogs, can address this need for informal tools. Without sophisticated formal tool support and process rules, social media has revolutionized how humans create and curate knowledge artifacts online [129]. Thus, researchers have started to transfer ideas and tools from social media into professional software development tools and processes. In the following section, work related to the use of social media by software developers is reviewed.

2.2 Use of Social Media by Software Developers

The need to have tool support for formal and informal activities is well recognized in software engineering [103]. Today's developers frequently make use of social media, also referred to as Web 2.0, to augment tools in their development environments [6]. That today's developers use social media to support collaborative development is not surprising as the computer industry is currently witnessing a paradigm shift in how everyday users work, communicate, and play over the Internet. This paradigm shift is due to the decentralization of computer systems and due to the wide array of innovative social media tools that are being adopted and adapted by this new generation of users. For example, social media applications, such as Facebook and Twitter, are household names.

Social media tools can be characterized by an underlying "architecture of participation" that supports crowd-sourcing as well as a many-to-many broadcast mechanism [129]. Their design supports and promotes collaboration, often as a side effect of individual activities, and furthermore democratizes who participates in activities that were previously in the control of just a few stakeholders.

Software engineers make use of a variety of social media tools to coordinate with one another, to communicate with and learn from users, to become informed about new technologies, and to create informal documentation. Despite the apparent widespread use of these technologies within software engineering, there is little known about the benefits or risks of using such tools, and the impact they may have on the quality of software. There is little advice on how, or indeed if, social media features should be integrated within modern software development environments.

In this section, social development environments are reviewed, as they provide the platform into which social media mechanisms could be integrated (Section 2.2.1). Subsequently, in Sections 2.2.2 through 2.2.7, related work on the use of particular social media mechanisms by software developers is summarized.

2.2.1 Social Development Environments

To support individual and team-based development activities, software engineers can choose from a wide variety of tools ranging from code editors and task management systems to integrated development environments and web-based portals for hosting projects. Over the last decade, the focus of tools has shifted from integrated development environments to collaborative environments, software project portals and forges. Here, this spectrum of tools is reviewed from a collaboration perspective.

Integrated Development Environments

Integrated Development Environments (IDEs) were initially designed as soloist tools, with features for the lone engineer to author, debug, refactor, and reuse code. Larger scale software projects are team based and require management of the artifacts under development. Thus most IDEs have data and control integration mechanisms to support team-based activities through version control, release management, and task management systems. In addition to these tools, mailing lists, forums, and wikis are commonly used to manage collaboration. These integrations predominantly rely on the Internet or the intranet as a platform, and they are critical in managing task articulation work during distributed development. Task articulation refers to how engineering tasks are identified, assigned and coordinated [72]. This kind of tool support was one the enablers of the current growth of global software development.

Collaborative Development Environments

Recently, the design of IDEs is leaning towards a federation of tools, as well as the addition of collaboration features [30]. IDEs that are designed with collaboration as a major focus are referred to as Collaborative Development Environments (CDEs) [105]. The primary goal of CDEs is to reduce friction in collaborative processes. IBM's Jazz, one example of a CDE, supports the integration of tools along the project lifecycle. The Jazz environment furthermore provides a web interface for accessing collaboration information, such as task management, and a web interface for customizing dashboards to provide information on project status [174]. This web interface extension of the CDE helps facilitate community involvement as they do not need to use the full blown IDE to browse and augment project information. The Microsoft Team Foundation Server has similar features to Jazz, such as task management and a team project portal to support collaboration[1].

Software Project Portals and Forges

Web-based portals that support software development activities have recently gained traction, and are continuing to change the way software is developed. Some of these portals are specifically designed for hosting projects, while others are for exchanging information or community building. Source code project forges host independent projects on a website. Software forge applications provide integrated web interfaces and tools, such as forums, mailing lists, wikis, bug trackers, and social networking, for accessing and managing the stored projects in a collaborative manner. Examples include SourceForge and Google Code.

Forges may have specialized software for setting up the project, with services for downloading the archives, and services for setting up and maintaining mailing lists, forums, wikis, and bug trackers. Forges are very popular for open source projects, and lately integrate more and more social media features. For example, Github[2] markets itself as a "social coding environment", because of its underlying philosophy in supporting social interactions around the project. Github combines social networking with the Git distributed source control; developers can follow other developers and they can watch specific projects, for example via "activity streams" [40].

There are other innovative websites that support developers in exchanging information and managing collaborative work. Stack Overflow is a community site where developers can ask and answer each others' questions. TopCoder[3] hosts programming competitions with the underlying business goal to connect companies with talented programmers. More recently, it is acting as an outsourcing service, where companies can assign challenging tasks to TopCoder programmers. Freshmeat[4] helps users keep track of software releases and updates, and hosts reviews and articles about software projects.

Some websites further integrate source code development features, such as authoring and compiling, within project websites. An interesting example is Skywriter[5], Mozilla's experiment with a HTML5-based code editor running entirely in a browser. Skywriter offers support for versioning and following of co-developers. The move of

[1]http://msdn.microsoft.com/en-us/teamsystem/
[2]http://www.github.com/
[3]http://www.topcoder.com/
[4]http://freshmeat.net/
[5]http://mozillalabs.com/skywriter (formerly named Bespin)

the IDE towards the browser further opens up opportunities for integration of social features [184].

These social development websites share the common theme of providing support for development activities using web-based social mechanisms. The term Social Development Environment (SDE) is used to refer to this broad spectrum of websites.

The use of social media cross-cuts many of the traditional categories in software development: it goes across teams, projects, and communities, and integrates a wide range of development processes and tools from IDEs to CDEs and SDEs. Social media usage can support software development activities ranging from requirements engineering and development to testing and documentation. The informal nature of social media channels allows developers to adapt them to their current context and has the potential to revolutionize the way collaborative software development is done. We are starting to see social media features becoming adopted and either used as an adjunct, or being directly integrated in the development environments [163].

In the following, related work on the use of social media by software developers is reviewed. Related work was identified through targeted web searches, by following the references in the papers found, and by using the knowledge gained during the review described previously.

2.2.2 Wikis

Wikis were designed with development collaboration directly in mind [111]. Since wikis have been around for some time, their adoption in software engineering is widespread[6]. One of the first articles describing the advantages and disadvantages of wikis in software development was authored by Louridas [116]. He explains that wikis offer flexibility that relieves project managers from having to produce a perfect document up-front. Wikis are easy to change, but due to their flexibility, they require tolerance and openness in the organization that they are implemented in.

So far, the use of wikis by software developers has been studied in four main areas: documentation, requirements engineering, collaboration, and knowledge sharing.

Wikis cater to many of the patterns for consistent software documentation presented by Correia *et al.* [38]: information proximity, co-evolution, domain-structured information, and integrated environments. One major advantage of using wikis for

[6]Another reason for the popularity of wikis in software development might be the fact that wikis were invented by the well-known software developer Ward Cunningham.

documentation is the possibility to combine different kinds of content into a single document. For example, Aguiar *et al.* introduce XSDoc wiki [5] to enable the combination of different kinds of software artifacts in a wiki structure. However, in a recent paper, Dagenais and Robillard found that open source developers who originally selected a public wiki to host their documentation eventually moved to a more controlled infrastructure because of maintenance costs and decrease of authoritativeness [42].

The flexibility of wikis is well-suited for a process with as much uncertainty as requirements engineering [48]. Lohmann *et al.* present a wiki-based platform that implements several community-oriented features aimed at engaging a larger group of stakeholders in collecting, discussing, developing, and structuring software requirements [115]. Ferreira and da Silva also present an approach for enhancing requirements engineering through wikis. Their approach focuses on the integration of a wiki into other software development tools [60]. Semantic wikis that enhance traditional wikis with ontologies offer the opportunity to further process wiki content. For example, Liang *et al.* present an approach for requirements reasoning using semantic wikis [112].

Al-asmari and Yu report on early experiences that show how wikis can support various collaboration tasks. They describe that wikis are easy to use, reliable, and inexpensive compared to other communication methods [7]. Several wiki-based tools aimed at supporting collaboration in software development have been proposed. Bauer *et al.* present WikiDev 2.0, a wiki implementation that integrates information about various artifacts, clusters them, and presents views that cross tool boundaries. Annoki, a tool from the same research group presented by Tansey and Stroulia [166], supports collaboration by improving the organization, access, creation, and display of content stored on the wiki. Galaxy Wiki presented by Xiao *et al.* takes the integration of wikis into software development processes a step further [193]: Developers write source code in the form of a wiki page and are able to compile, execute, and debug programs in wiki pages. Wikigramming by Hattori follows a similar approach [83]: Each wiki page contains a Scheme function which is executed on a server. Users can edit any function at any time, and see the results of their changes right away.

Closely related to the use of wikis for collaboration is their use for knowledge sharing. Phuwanartnurak and Hendry report preliminary findings on information sharing that they discovered through the analysis of wiki logs [139]. They found that many wiki pages were co-authored, and that wiki participation differs by role. Clerc *et al.* found that wikis are particularly good for managing and sharing architectural knowledge [35]. Another paper by Rech *et al.* describes that the knowledge sharing enabled by wikis allows for increased reuse of software artifacts of different kinds [141]. Correia *et al.* propose Weaki, a weakly-typed wiki prototype designed to support incremental formalization of structured knowledge in software development [39]. Semantic wikis also play a role in knowledge sharing: Decker *et al.* propose support for structuring the knowledge on wikis in form of semantic wikis [49]. A more recent paper by Maalej *et al.* provides a survey of the state-of-the-art of semantic wikis [118]. The authors conclude that semantic wikis provide lightweight, incremental, and machine-readable software knowledge articulation and sharing facilities. Further applications of wikis in software engineering include traceability and rationale management as described by Geisser *et al.* [71].

2.2.3 Blogs

The role of blogs in software development is not as well understood as the role of wikis [133]. Some practitioners advocate that every developer should have a blog[7], arguing that blog posts help exchange technical knowledge among a larger audience than email messages. A comprehensive study of the use of blogs by software developers to date was conducted by Pagano and Maalej [132]. The authors found that the most popular topics that developers blog about are high-level concepts, such as functional requirements and domain concepts, and that developers are more likely to blog after corrective engineering and management activities than after forward engineering and re-engineering. An earlier study from the same research group by Maalej and Happel looks at how software developers describe their work [117]. They found that blogs and other media are a step towards diaries for software developers.

The theme of requirements engineering through blogs was also addressed by Park and Maurer [133]. They discuss strategies for gathering requirements information through blogs. However, as observed by Seyff *et al.*, end-user involvement in software development is an ambivalent topic [156]. They present a mobile requirements elicitation tool that enables end-users to blog their requirements in situ without facilitation by analysts.

In many companies, blogging has found its way into corporate culture outside

[7]http://channel9.msdn.com/posts/Glucose/Hanselminutes-on-9-Social-Networking-for-Developers-Part-1-Every-Developer-Needs-a-Blog/

of a software engineering context. In a study from 2007, Efimova and Grudin [55] describe emergent blogging practices in a corporate setting. They found blogging in the enterprise to be an experimental, rapidly evolving terrain in which balancing personal and corporate incentives and issues is one of the challenges that bloggers face. Huh *et al.* [95] conducted a similar study and found that the corporate blogging community allows access to tacit knowledge and that it contributes to new forms of collaboration within the enterprise.

2.2.4 Microblogs

The use of microblogging services, such as Twitter, by software developers was first studied by Black *et al.* [22]. Most respondents in their survey reported using several social media tools, and Twitter as well as instant messaging were among the most popular tools. A more detailed study on the use of Twitter by software developers was authored by Bougie *et al.* [24]. They found that software developers use Twitter's capabilities for conversation and information sharing extensively and that the use of Twitter by software developers notably differs between users from different projects.

Several researchers have started to integrate microblogging services into development environments. Reinhardt presents an Eclipse-based prototype for integrating Twitter and other microblogging services into the IDE [142]. Adinda, a browser-based IDE introduced by van Deursen *et al.* [184], includes microblogging for traceability purposes, in particular aimed at tracking which requirements are responsible for particular design decisions, code fragments, and test cases. A more detailed description of the microblogging integration envisioned by this research group is given by Guzzi *et al.* [78]. They present an approach that combines microblogs with interaction data collected from the IDE. When they evaluated their approach, participants in their study used the microblogging tool to communicate future intentions, ongoing activities, reports about the past, comments, and future tasks.

2.2.5 Tags

The concept of annotating resources using tags is not new to software development. Hassan and Holt presented an approach that recovers information from source control systems and attaches this information to the static dependency graph of a software system [82]. They refer to this attached information as source sticky notes, and they showed that the sticky notes can help developers understand the architecture of large software systems. A complimentary approach that employs annotations edited by humans rather than automatically generated annotations was presented by Brühlmann *et al.* [28]. They proposed a generic approach to capture informal human knowledge in form of annotations during the reverse engineering process. Annotation types in their tool Metanool can be iteratively defined, refined, and transformed, without requiring a fixed meta-model to be defined in advance. This strength of annotations – the ability to refine them iteratively – is also employed by the tool BITKit presented by Ossher *et al.* [130]. In BITKit, tags are used to identify and organize concerns during pre-requirements analysis. The resulting tag structures can then be hardened into classifications to capture important concerns.

There is a considerable body of work on the use of annotations in source code. The use of task annotations in Java source code, such as TODO, FIXME, or HACK, was studied by Storey *et al.* [161]. They conducted a study that explored how task annotations embedded within the source code play a role in how software developers manage personal and team tasks. Data was collected by combining results from a survey of professional software developers, an analysis of code from open source projects, and interviews with software developers. The authors found that task management is negotiated between the more formal modification request systems and the informal annotations that developers add to their source code.

Based on this research, the tool TagSEA was developed, a collaborative tool to support asynchronous software development [160]. The authors' goal was to develop a source code annotation tool that enhances navigation, coordination, and the capture of knowledge relevant to a software development team. The design was inspired by combining waypoints from geographical navigation with social tagging from social bookmarking software to support coordination and communication among software developers. TagSEA was evaluated in two longitudinal empirical studies that indicated that TagSEA was used to support reminding and refinding [162]. TagSEA was extended to include a Tours feature that allows programmers to give live technical presentations that combine static slides with dynamic content based on TagSEA annotations in the IDE [33].

TagSEA has been applied by other researchers for more advanced uses. Boucher *et al.* described how they used tagging to identify features in source code [23]. These tags are then used to prune source code for a pragmatic approach to software product line management. In eMoose, presented by Dekel and Herbsleb [50], developers can associate annotations or tag directives within API documentation. eMoose then pushes these annotations to the context of the invoking code by decorating the calls and augmenting the hover mechanism.

Closely related to the concept of tagging is the social bookmarking concept, a method for organizing, storing, managing, and searching bookmarks online. Such bookmarks are often organized using tags. Guzzi *et al.* present Pollicino, a tool for collective code bookmarking. Their implementation was based on requirements gathered through an online survey and interviews with software developers. A user study found that Pollicino can be effectively used to document developers' findings that can be used by other developers [77].

2.2.6 Feeds

Feeds in software development are used to achieve awareness in collaborative development settings, and information overload quickly becomes a problem [174]. Fritz proposed an approach that integrates dynamic and static information in a development environment to allow developers to continuously monitor the relevant information in context of their work [64]. In a later paper, Fritz and Murphy give more details on their concept [66]. Through a series of interviews, they identified four factors that help developers determine relevancy of events in feeds: content, target of content, relation with the creator, and previous interactions.

Calefato *et al.* proposed a tool to embed social network information about distributed co-workers into IBM's Jazz [30]. They used the FriendFeed aggregator to enhance the current task focused feed implementation with additional awareness information to foster the building of organizational values, attitudes, and inter-personal relationships.

2.2.7 Social Networks

Concepts from social networking sites, such as Facebook, have been brought into software development through the Codebook project by Begel *et al.* [12]. Codebook, based on earlier work by Venolia [186] and work on Deep Intellisense [92], is a social networking service in which developers can be friends not only with other developers but also with work artifacts. These connections then allow developers to keep track of task dependencies and to discover and understand connections in source code. Codebook was inspired by a survey with developers that found that the most impactful problems concerned finding and keeping track of other developers [14]. Based on the Codebook framework, Begel *et al.* introduced a newsfeed [16] and a search portal [15].

2.2.8 Summary

Several social media mechanisms, such as wikis, blogs, and tags, have been adopted by software developers in various different contexts, and researchers have developed a number of tools aimed at transferring ideas from successful social media applications, such as Facebook and Twitter, into professional software development. However, there is a lack of research that investigates the role that social media can play in software development, what differentiates social media artifacts from other development artifacts, and how social media mechanisms can interplay with more traditional mechanisms used by software developers. This thesis takes a broader perspective by examining the role of social media artifacts in terms of their impact on specific software development processes, such as task management and documentation, rather than narrowly focusing on particular tool features.

In the following part, the findings on task management are reported.

Part II

Task Management

Chapter 3

Task Management using IBM's Jazz

In this chapter, the findings on the role of social media artifacts in task management are described as they relate to software development teams using IBM's Jazz. Jazz is an extensible technology platform that helps teams integrate tasks across the software lifecycle. The software development team collaboration tool built on top of Jazz is called Rational Team Concert (RTC)[1]. Jazz was one of the first environments to tightly integrate social media artifacts, such as tags, dashboards, and feeds, into an IDE.

First, Section 3.1 describes how tagging, a social media mechanism, is used to communicate matters of concern in the management of development tasks. The results stem from two empirical studies over 36 and 12 months respectively on how tagging has been adopted and what role it plays in the development processes of several professional development projects with more than 1,000 developers in total. These studies show that the tagging mechanism was eagerly adopted by the teams, and that it has become a significant part of many informal processes. Different kinds of tags are used by various stakeholders to categorize and organize tasks. The tags are used to support finding of tasks, articulation work, and information exchange. Implicit and explicit mechanisms have evolved to manage the tag vocabulary. These findings indicate that tool support inspired by social media can play an important role in improving team-based software development practices.

Section 3.2 describes a tool developed based on these findings, called CONCERN-

[1]https://jazz.net/products/rational-team-concert/

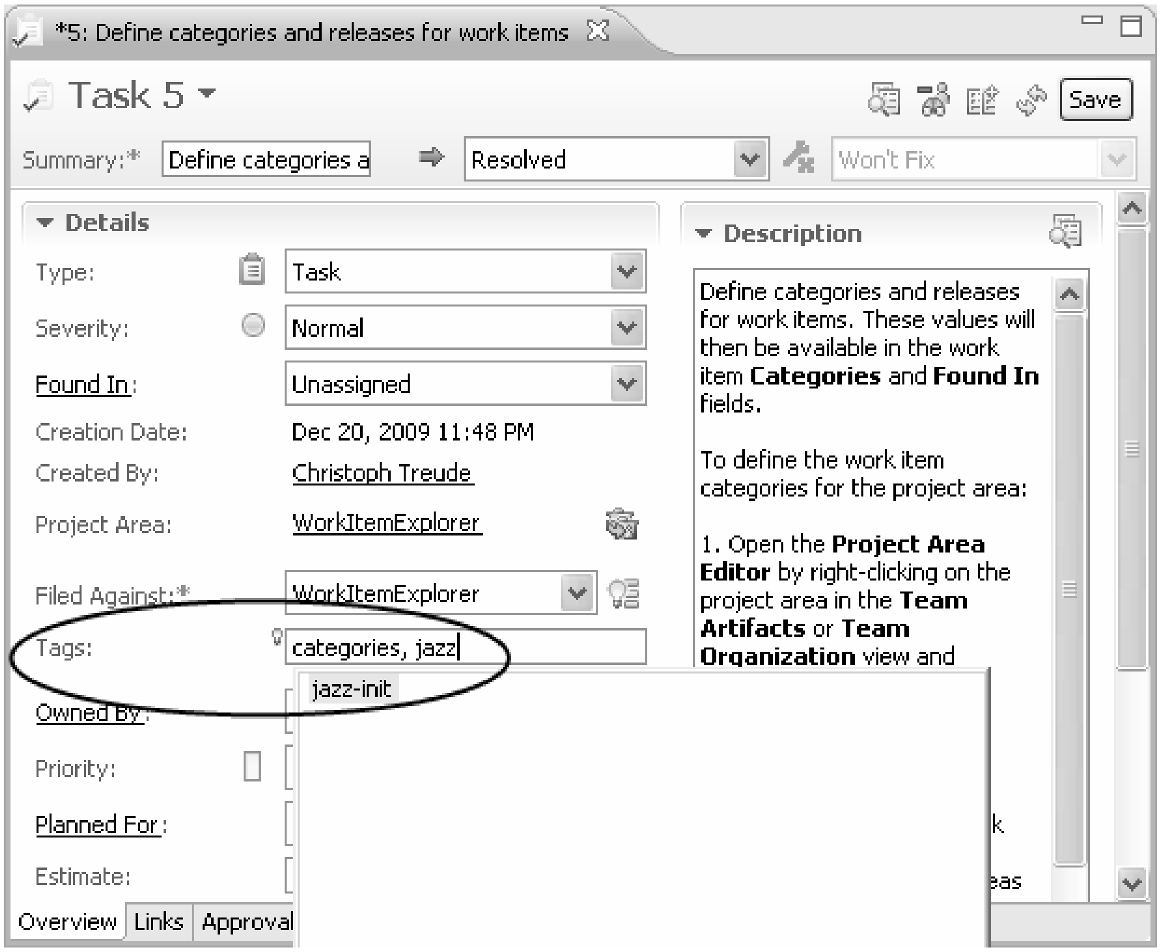
Lines. ConcernLines supports the cognitive process of understanding how information about the release history, non-functional requirements, and project milestones relates to functional requirements on software components by visualizing co-occurring concerns over time.

3.1 Tagging in IBM's Jazz

Software development environments typically have explicit tool support for managing tasks. For example, IBM's Jazz has tool support for managing "work items", where a work item is a generalized notion of a development task (see Figure 3.1). Work items are assigned to developers, are classified using pre-defined categories, and may be associated with other work items. Jazz work items also have tool support to address social aspects. Specifically, Jazz supports a discussion thread and a "tagging" mechanism. Using this latter feature, developers can freely associate user-defined keywords with work items.

Here, results from two empirical studies on the practice of tagging work items within Jazz are reported. The case studies examine how industrial software development teams use tags for work items, both for individual use and for collaboration. Data was gathered through the inspection of the project repositories and by conducting interviews with developers on different teams. The main contribution in this chapter is the identification of different ways in which tags support informal processes in software development by filling the gap between social and technical aspects of artifacts. I explore how tagging supports collaboration in various ways. Furthermore, I examine how tagging was adapted by software developers to suit their needs, and I identify potential tool enhancements.

The remainder of this section is structured as follows. In Section 3.1.1, related work on tagging is discussed. The tagging feature in IBM's Jazz is introduced in detail in Section 3.1.2. The research questions and methodology are presented in Section 3.1.3. Section 3.1.4 comprises the main part of this research and describes how tagging of work items has been adopted, what role it plays in software development processes, how it supports collaboration, and how tool support for tagging can be improved. The limitations of the studies are presented in Section 3.1.6, and the findings are summarized in Section 3.1.7.



*5: Define categories and releases for work items

Task 5

Save

Summary:* Define categories a

Resolved

Won't Fix

Details

Type: Task

Severity: Normal

Found In: Unassigned

Creation Date: Dec 20, 2009 11:48 PM

Created By: Christoph Treude

Project Area: WorkItemExplorer

Filed Against:* WorkItemExplorer

Tags: categories, jazz

jazz-init

Owned By:

Priority:

Planned For:

Estimate:

Overview | Links | Approval

Description

Define categories and releases for work items. These values will then be available in the work item **Categories** and **Found In** fields.

To define the work item categories for the project area:

1. Open the **Project Area Editor** by right-clicking on the project area in the **Team Artifacts** or **Team Organization** view and

Figure 3.1: Work item interface in IBM's Jazz

3.1.1 Related Work: Tagging and Software Development

The concept of tagging originates from the social computing domain. Social computing technologies, sometimes referred to as Web 2.0, are seeing rapid adoption by emergent communities on the web. Key examples include Facebook[2], YouTube[3] as well as community-based recommender systems, such as CiteULike[4], TripAdvisor[5], and Flickr. Tagging is used by many of these systems and is often referred to as social bookmarking. The success of tags is closely related to their bottom-up nature: tags do not have to be pre-defined, every user can choose their own tags, and the number of tags per item is arbitrary. Based on these characteristics, tags are used to classify

[2]http://www.facebook.com/
[3]http://www.youtube.com/
[4]http://www.citeulike.org/
[5]http://www.tripadvisor.com/

items in an informal way, and they stand in contrast to formal top-down classification mechanisms.

A tagging system consists of three main components [89, 147]: tag users, the tags themselves, and the objects being tagged. In most social tagging systems that have been studied thus far, the items being tagged are often heterogeneous and may come from a very large pool of uncontrolled resources. The typically large number of creators and users of the tags tend to be from a very large uncontrolled population, with varying levels of expertise. Most systems keep track of who tagged which object, useful metadata which can be used to infer the interests of a particular user as well as count how many times a given tag is assigned to an object (thus providing a way to reinforce the relevance of tags assigned).

Golder *et al.* [74] and Hammond *et al.* [81] provide overviews of tagging systems and classify the main reasons for tagging. A common finding across these studies is that users tag to provide information on an artifact (e.g., what an artifact is or to refine a category) and for organizing artifacts. A more detailed study was conducted with the photo sharing website Flickr [9]. Robu *et al.* [147] examined data from Delicious[6], a social bookmarking site, to describe the dynamics of collaborative tagging systems. Their findings indicate that despite the unsupervised tagging by individual users, coherent and rich categorization schemes emerge especially for specialized domains.

Heymann *et al.* analyzed the social tagging of books and found that the tagging system was fairly consistent, of high quality, and complete [89]. Sen *et al.* [154] explored how tagging is used by a community using the MovieLens recommender system. Sen's research questions focused on how personal tendencies and community influence the creation of tags. Part of the success of tagging comes from allowing users to define their own vocabulary [69]. Information retrieval is also enhanced by community tagging [147].

The introduction of tags into software development raises the question of how the informality of tagging affects the process of developing software and how a typical software development process can take advantage of the characteristics of tags. Tagging is not a new concept to software engineering; however, earlier forms of tagging are not consistent with the social computing notion of tagging today [27]. Many early uses of the word tagging in software engineering systems relied on a pre-existing controlled vocabulary. Tags have been used for decades for annotating check-in and branching events in software version control systems, as well as for documenting bugs

[6]http://delicious.com/

in bug tracking systems. Also, Brothers' ICICLE was an early exploration of tag-like structures with a limited, controlled vocabulary during code inspection [27].

Due to these inconsistencies on the term tagging, I define a *tag* as follows:

> A tag is a freely-chosen keyword or term that is associated with or assigned to a piece of information. In the context of software development, tags are used to annotate resources, such as source files or test cases, in order to support the process of finding or reporting on these resources. Multiple tags can be assigned to one resource.

The term **tag keyword** is used to indicate the term that is used (e.g., `usability`), and the term **tag instance** is used to indicate instances of a tag keyword being applied to one or more resources.

Marlow *et al.* [122] give a detailed taxonomy of tagging systems by defining seven dimensions pertinent to the design of a tagging system:

- *Tagging rights:* Users can tag everybody's resources vs. users can only tag their own resources.
- *Tagging support:* Blind tagging (users cannot see each other's tags) vs. viewable tagging (users can see each other's tags) vs. suggestive tagging (the system suggests tags to users).
- *Aggregation:* Bag model (allows duplicate tags per resource) vs. set model (no duplicates).
- *Type of object:* Type of the resource to be tagged.
- *Source of material:* Users need to upload the resources to be tagged vs. resources are supplied by the system.
- *Resource connectivity:* Different resources can have links to each other vs. different resources can be grouped together vs. no connections between resources.
- *Social connectivity:* Different users can have links to each other vs. different users can be grouped together vs. no connections between users.

While these dimensions apply to tagging systems used by software developers, studying tagging systems used by software developers, such as ICICLE [27], TagSEA [162], IBM's Jazz, BITKit [130], Google Code, ConcernMapper [145], and Concern Graphs [144], reveals additional dimensions on top of Marlow's taxonomy:

- *Pre-defined vs. user-defined:* Most current tagging systems are based on the concept of tags as freely-chosen keywords or terms that are associated with or assigned to a piece of information. However, in older tagging systems, such as ICICLE [27], possible keywords were pre-defined, and software developers were not able to add new keywords to the system.

- *Metadata:* Different tagging systems store different amounts of metadata. For example, in the case of tagging work items in IBM's Jazz, information such as the tag author and the time a tag was applied to a work item, can only be identified by browsing the work item's history. In other systems, such as TagSEA, the author and time can be explicitly added to each tag instance, and tags can be searched by their authors and creation time.

- *Semantics:* While most tagging systems treat keywords simply as terms that are associated with artifacts, some systems go beyond that and add meaning to certain tags or tags with certain characteristics. In that case, some keywords or some keyword characteristics have to be pre-defined. An interesting approach is taken by labels in the Google Code Issue Tracker[7], which goes beyond basic labels to support key-value labels. Key-value labels contain one or more dashes, and the part before the first dash is considered to be a field name while the part after that dash is considered to be the value.

- *Hierarchies:* Some tagging systems explicitly support tag hierarchies, using a dot-notation (e.g., TagSEA). Keywords that have dots in them can be treated as hierarchical, and they can be displayed in tree-views. In other systems, such as IBM's Jazz, some developers use the dot-notation even though there is no explicit support for hierarchies.

- *Resource type:* Software developers handle many different kinds of artifacts from source code and work items to build scripts. Nevertheless, many tagging systems for software developers only support tagging of a single kind of artifact. One exception is TagSEA. It allows software developers to tag locations in source code – called waypoints – and artifacts, such as files, and it shows different kinds of artifacts in a single view. This allows for grouping and relating different kinds of artifacts while keeping the simplicity of tags.

[7]http://code.google.com/p/support/wiki/IssueTracker#Labels

- *Integration:* Another dimension is the extent to which the tagging mechanism is integrated with other tooling. Some systems support social tagging of source code, but require the user to post code fragments on public servers before tags can be applied to code fragments (e.g., DZone Snippets[8] and ByteMyCode[9]). In other systems, such as IBM's Jazz or TagSEA, the tagging mechanism is part of the IDE.

So far, there is little research on how the social media mechanism of tagging can play a role in supporting activities in software development. The research described in this section examines the current use of tags in software development projects with the aim to identify their role in task management.

3.1.2 Work Item Tags in IBM's Jazz

Developers using IBM's Jazz organize their work around so-called work items which can be interpreted as development tasks. A typical work item as shown in Figure 3.1 consists of a unique number, summary, description, state, work item type, severity and priority, the component it was filed against, the version it was found in, the creator, and several other details that are optional. The primary way to organize work items in Jazz is to use the category hierarchy. The category for each work item is identified by filling out the *Filed Against* field. A work item can only be in one category and the available categories are defined per project by the development manager.

As can be seen in Figure 3.1, there is an optional tag field in which developers can insert an arbitrary number of tag instances per work item. The Jazz content assistant suggests tag keywords with a common prefix that have been used before. If a developer adds a tag keyword that has not been used before, a pop-up window appears and asks if this keyword should be added to the vocabulary. Tag instances are public to all members of a project team across all components.

Compared to online media tagging and social bookmarking, tagging of work items in Jazz is different in some important aspects. In Jazz, the items being tagged are strictly homogeneous work items that are created by members of the Jazz community. Typically creators of Jazz work items and tags have some expertise in the underlying

[8]http://snippets.dzone.com/
[9]http://bytemycode.com/

software project and would not be classed as casual users. Another potentially important difference is in terms of the metadata associated with the tag instances and keywords. As mentioned above, most tagging systems record the user that attached a particular tag instance to a resource. In Jazz, the tag instances are added directly to a work item and information on when the tag instance was added and by whom is not easily accessible (only through the work item's history). Tag instances can also be removed from a work item by anybody with access to that work item (typically all developers on a project). A tag instance can only be attached "once" to a given work item, and the creator of the tag instance is not visible in the default view.

Thus, the findings on tagging in IBM's Jazz may be expected to be somewhat different from the existing results in this area. The research questions that explore how the Jazz tagging feature supports collaborative software development are listed in the next section.

3.1.3 Research Methodology

This section presents the research questions related to the role of work item tags in IBM's Jazz as well as the research setting and the methods that were used for data collection and analysis.

Research Questions

The research questions examine the tagging activity from three perspectives: adoption, characteristics, and the role of tagging.

1. How is the social tagging mechanism adopted by developers for annotating work items?
   (a) How does the frequency of new tag instances vary over the lifetime of a project?
   (b) How many work items are tagged?
   (c) How many users tag and how does this number vary over time?
2. What characteristics of tags are prevalent in the tagging of work items?
   (a) Which tag keywords are applied more frequently?

   (b) What are the different categories of tag keywords that emerge during a project?

3. What role does the tagging feature play in the work practices of individual and team developers?

   (a) Are work item tags intended for individual and/or collaborative use?

   (b) Why do developers tag work items?

   (c) How do developers use tags?

   (d) How are tags managed and why are tag instances removed?

   (e) How does a team reach consensus on the tag vocabulary?

Research Setting

The studies took place with several professional development teams from IBM, as described in the following paragraphs.

Case Study 1: Jazz. The first case study was conducted with the Jazz development team. The team consists of approximately 175 contributors and about 30 functional teams, with some teams acting as sub-teams of larger teams and some contributors assigned to multiple teams. The team members are located at 15 locations worldwide, primarily in North America and Europe. The developers of the team have been self-hosting their development since early 2006, and they follow the "Eclipse Way" development process [68]. This process, developed by the Eclipse Development Team, is an agile, iteration-based process with a focus on consistent, on-time delivery of quality software through continuous integration, testing, milestones, and incremental planning. At the time of data collection, the developers were working on the 2.0 release of RTC, and they were using the latest milestone builds of RTC for their development.

Case Study 2: Enterprise Infrastructure (EI). The study was replicated with a large project team[10] of more than 1,000 members working on four interrelated projects. They had been using Jazz for about one year and had developed systems

[10]The project name is obfuscated for confidentiality reasons.

mostly for enterprises. They were part of IBM, but not connected to the Jazz development team. The development processes used by these teams range from Scrum to conventional methods. At the time of this study, the teams were using RTC 1.0.

Data Collection

The methodology follows a mixed-methods approach, collecting quantitative and qualitative data. In order to gather quantitative data on the use of tags in the project, the repositories of the development teams were accessed and all relevant information was extracted. The amounts of data extracted for both case studies are shown in Table 3.1.

Table 3.1: Tag related data extracted from repositories

case	data object	amount
Jazz	Work items	65,268
	Tag instances applied to work items	27,252
	Tag instances removed from work items	2,452
	Number of unique tag keywords	1,184
EI	Work items	50,826
	Tag instances applied to work items	13,588
	Tag instances removed from work items	758
	Number of unique tag keywords	673

Qualitative data was collected through a series of interviews with developers and through ethnographic-style observations. All interviews were semi-structured allowing for follow-up questions and clarifications. Most of the questions were aimed at understanding the details of why and how developers use tags. Appendix A.1 shows the list of questions used to guide the interviews. In total, twelve interviews were conducted: six for each case study. For the Jazz case study, I interviewed the development manager **J-M**, the project administrator **J-A**, one component lead **J-L1**, and three developers on one team **J-D1**, **J-D2**, and **J-D3**. For the EI case study, I interviewed one product design lead **EI-L**, a development manager **EI-M**, a release engineer **EI-R**, and three developers who occasionally took the role of scrum master **EI-D1**, **EI-D2**, and **EI-D3**. All interviews were conducted in-person at an IBM location and lasted about 30 minutes each.

In addition, I spent seven months at the Jazz site and two weeks at the EI site as part of an ethnographic study. I frequently had informal discussions with developers regarding their use of tags and the answers in the interviews were mirrored in my observations. The observations were recorded using ethnographic field notes. The quantitative nature of the repository analysis and the qualitative nature of the interviews and observations provided insights for all of the previously posed research questions.

Data Analysis

A Jazz plugin was developed to extract the data related to tags from the repositories of all development teams in the studies. The pertinent data extracted contained all work items along with their IDs, creators, creation times, owners, summaries, descriptions, priorities, severities, and several other fields. In addition, the following data was extracted for each instance of a developer applying a tag to a work item: the time that the tag instance was created, the tag keyword that was used, the time of creation, and the creator. Instances of tag keywords being removed from a work item were also extracted.

We collaboratively coded tag keywords in a group of two researchers to identify the categories of keywords that emerged over the duration of the projects. We coded the tag keywords individually first, and then confirmed our codes in several collaborative coding sessions in which we considered everything we know about each tag keyword before assigning codes to it. The coding was done using the bottom-up inductive technique of Corbin and Strauss [37]. Although there are some findings on the categories of tags for social bookmarking systems (as mentioned earlier), we expected to see very different categories emerge in this study of tagging and we did not start our coding process with an initial set of codes.

During the tag coding process, we considered all of the 12 interviews conducted, and we followed up on keywords that we were not able to categorize through email discussions with three of the participants (J-A, EI-R, and EI-D2). In addition, we read summaries and descriptions of the work items that were tagged with particular keywords to confirm our classification. For the Jazz project, we also accessed the project internal mailing lists as well as the documentation available on the project website. Based on these codes, we identified more abstract categories in which we grouped tag keywords using similar codes.

We also coded the interviews in collaborative sessions. For some of the research questions, such as “Why do developers tag work items?”, the answers we considered were mainly the answers to that particular question in the interviews. For other research questions, in particular the ones regarding the collaborative aspects of work item tagging, themes emerged through the assignment of codes to quotes and grouping of codes.

For each interview snippet, sometimes multiple codes would apply. We then grouped the quote segments and extracted the most prominent themes that appeared repeatedly in the interview data. When exploring the interview data, we made use of the tagging data to help us in the interpretation of the quotes. For quotes that were unclear, we checked with Jazz and EI team members on our understanding of their tagging processes. In our analysis of the tagging keywords and the interviews, the field notes from the ethnographic observations were crucial in helping us make sense of the data.

3.1.4 Findings

This section presents the findings, categorized by the research questions.

RQ1: Adoption of Tagging

To answer the first research question on the adoption of tags, an analysis of new tag instances over time was performed, looking at the number of tag instances that are applied to work items and the number of individuals tagging work items.

Frequency of New Tag Instances. Figure 3.2 show how the number of tag instances added per day evolves over time in both case studies. The grey line depicts the actual numbers per day; the black line gives the value averaged over the last 30 days at any point of time. The moving average line was added to allow for easier visual interpretation. The graphs are not significantly different when calculating the average for longer or shorter time intervals. For the Jazz project, the number increases until mid-2008, then drops, and increases again towards mid-2009. Mid-2008 and mid-2009 marked the two major releases of Jazz. Apart from high tagging activity in the beginning, the rolling average of tag instances in the EI case study is stable at around 40 instances per day. Spikes are mostly related to planning activities, such as coordinating which work items should be included in a particular release.

To see the extent to which the number of new tag instances depends on the number of new work items, the ratio of new tag instances to new work items per day

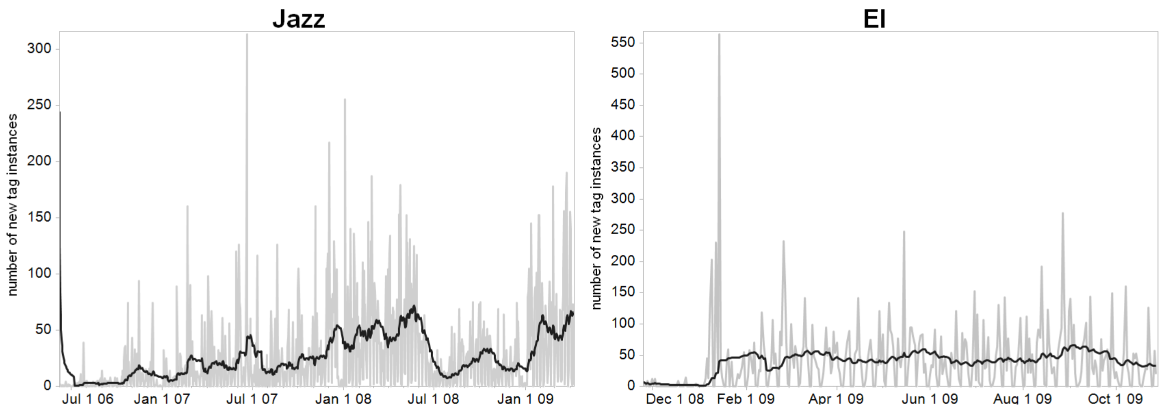


Figure 3.2: Number of new tag instances over time

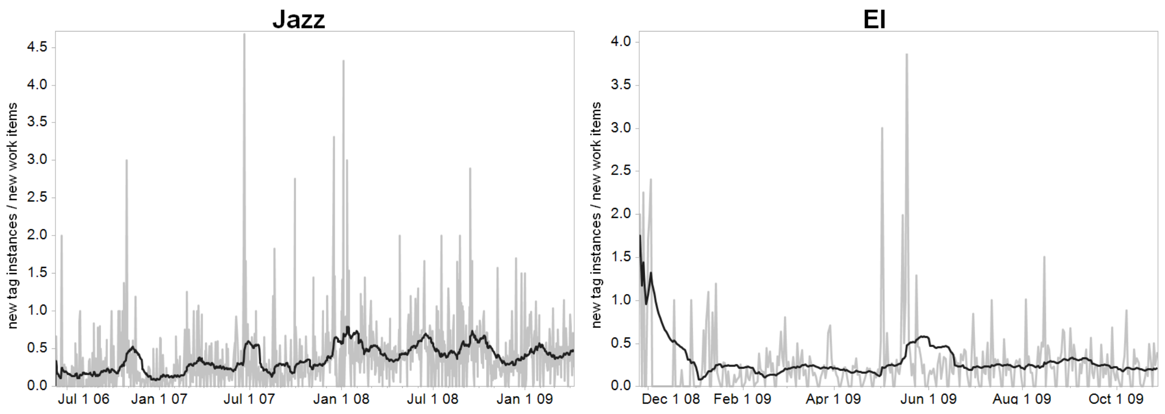


Figure 3.3: Rate of new tag instances vs. new work items over time

was calculated as shown in Figure 3.3[11]. For both case studies, the rate does not change substantially over time, apart from a few spikes.

Distribution of Tag Instances to Work Items. About 28.5% of all work items in Jazz and about 18.8% of all work items in EI have been tagged at least once. The distribution of tag instances to work items is shown in Figure 3.5 for Jazz using a log scale. The distribution of tag instances to work items for EI follows the same pattern.

Number of Users Applying Tag Instances. The number of individuals applying tag instances to work items over time follows a similar pattern as the number of tag instances. As shown in Figure 3.4, there are peaks of up to 40 different individuals applying tag instances on the same day in Jazz, and the only major discontinuities

[11]Undefined values resulting from a division by 0 on days with no new work items are represented as 0 for simplicity reasons.

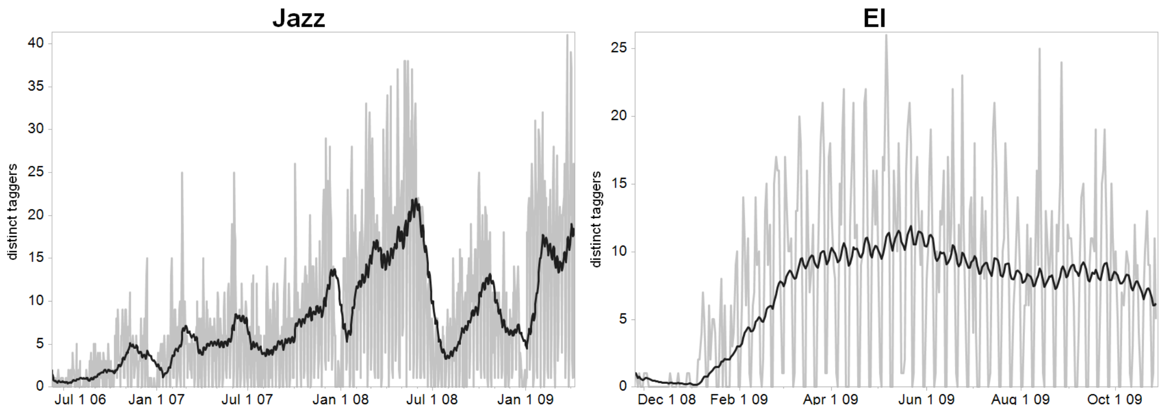


Figure 3.4: Number of distinct taggers over time

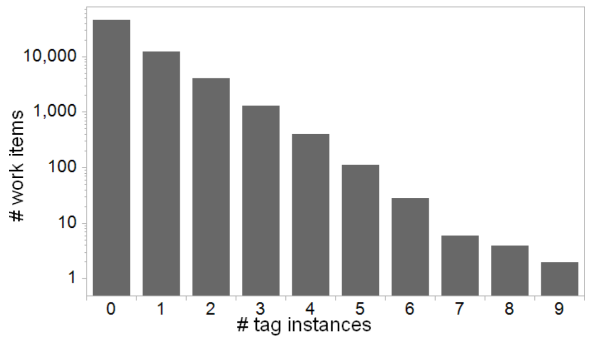


Figure 3.5: Distribution of tag instances to work items in IBM's Jazz

in distinct users per day occur around the Christmas holidays and after the release in mid-2008. For EI, the number of distinct users per day is between 5 and 10 on average, with peaks of up to 25 users.

In Jazz, 360 contributors have applied at least one tag instance to a work item. Out of 299 contributors who owned work items in the last three years, 176 (59%) applied at least one tag instance to a work item. In addition, the project has a web portal that allows clients to submit new work items. There were 184 individuals from outside the company applying tag instances through this web portal. However, the main tag users were team members from inside IBM. The top 50 most prolific taggers applied between 125 and up to more than 3,000 tag instances, using about 100 different keywords. In EI, 314 (29%) contributors have applied at least one tag instance to a work item. In total, the EI project has 1,082 members. The top 25 most prolific taggers applied between 150 and 800 tag instances, using about 50 different keywords.

These numbers indicate that tags were used continuously after their initial introduction and that software developers found them helpful enough to keep using them over a period of three years. More details on tag usage in support of informal processes and collaboration are given below.

RQ2: Characteristics of Tag Keywords

This section describes the characteristics of the tag keywords that the studies revealed. To address this topic we looked at the analysis of the tag keywords and instances as the primary data source, but also used interviews (and follow-up discussions) to explain and confirm the findings.

Most Frequently Applied Tag Keywords. In Jazz, 1,184 different keywords were applied in the time frame of the case study (May 2006 – April 2009). Table 3.2 shows the ten most frequently applied tag keywords in Jazz.

Table 3.2: Tag keywords with the most instances in Jazz

tag keyword	# instances
`polish`	966
`svt`	870
`ux`	668
`tvt`	636
`testing`	565
`globalization`	442
`usability`	441
`maintenancecandidate`	436
`errorhandling`	431
`mustfix`	421

In EI, 673 different keywords were applied in the time frame of the case study (November 2008 – October 2009). Table 3.3 shows the ten most frequently applied tag keywords. Italicized keywords are obfuscated for confidentiality reasons.

In both case studies, the distribution of instances to tag keywords has the shape of a "long tail". The long tail distribution has been frequently observed in tagging systems [147, 154].

Different Kinds of Tag Keywords. Our classification of the tag keywords from both case studies reveals that different kinds of tag keywords exist in the projects.

Table 3.3: Tag keywords with the most instances in EI

tag keyword	# instances
external library	601
product X	573
`id`	481
`teamb`	466
`a11y`	433
`docs`	427
product Y	420
component	403
`dev`	357
`conformance`	336

Unlike tagging in other applications, such as tagging of photos on Flickr, tags for work items do not necessarily describe the content of the tagged item. EI-R describes: *"If you look at* ***`dev`*** *and* ***`teamb`****, they're not that descriptive. And* ***`conformance`***[12] *wasn't really used for that – it doesn't really describe what the team itself is working on as opposed to the nature of that specific work item."* EI-D2 adds: *"They don't necessarily describe the content of the work item, they might link a bunch of [work items] together in a way that wouldn't be obvious just from looking at one work item, but because I know that they kind of all belong to the same initiative or the same project, they make sense that way. [...] Sometimes it's more meta information than the actual item itself."*

Figure 3.6 shows the results of our classification. We classified the keywords accounting for the "heads" of the "long tails" (i.e., all keywords that accounted for 80% of all tagging instances in the data). Thus, we classified the 197 most-used keywords from the Jazz data, and the 77 most-used keywords from the EI data. The keywords we classified had at least 28 instances each for the Jazz data, and at least 34 instances for the EI data.

We identified 10 categories that apply to both case studies as shown in Figure 3.6. We assigned one category to each tag keyword. Despite the ethnographic-style observations, reading the work items, interviews with 12 participants, and follow-up email discussions, there were tag keywords that we were unable to classify. Those keywords are shown in the right-most column – only 26 for the Jazz project, and 8 for the EI

[12]The keyword `conformance` was used to indicate work items related to supporting third party systems, such as browsers.

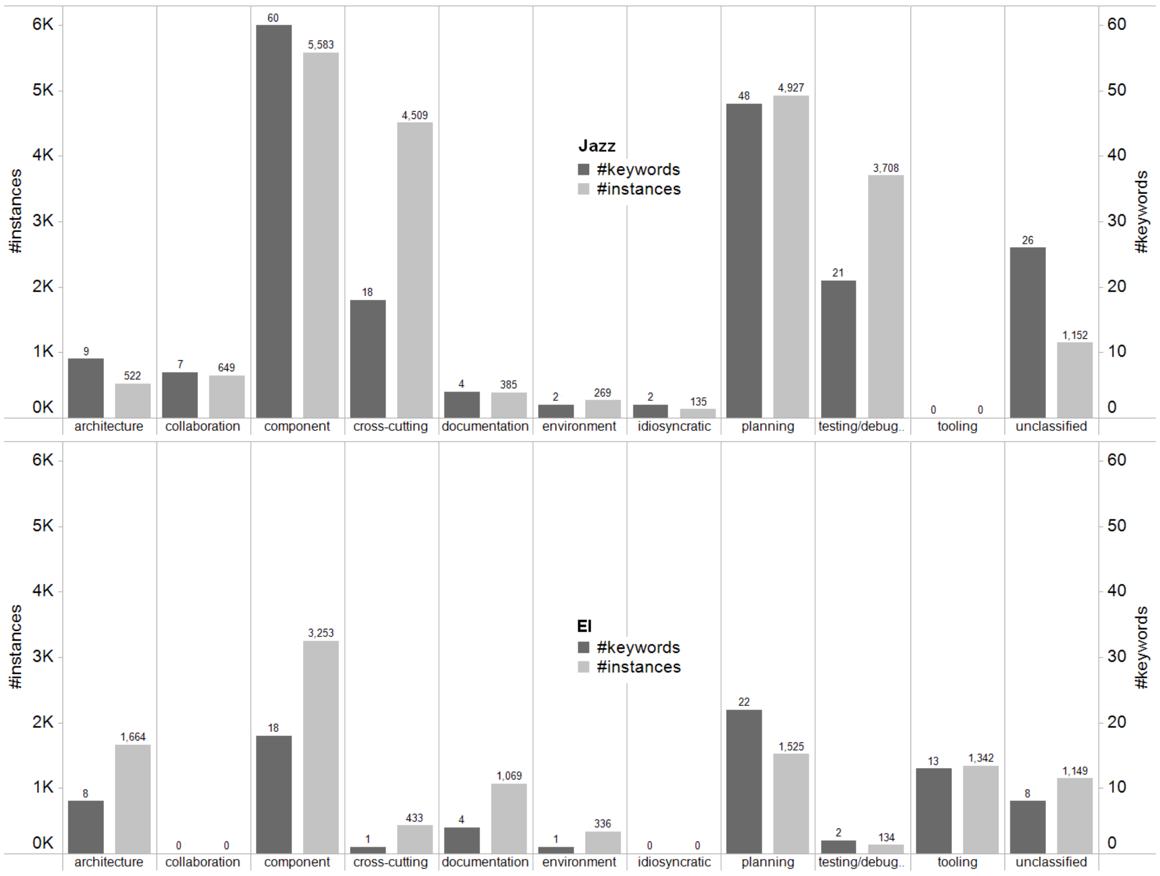


Figure 3.6: Number of tag keywords and instances per category

project. Each category of the classified tag keywords is described in the following paragraphs.

Architecture. In both data sets, tags were used to mark work items related to architecture. These work items were either about the overall architecture of the product or about integrating the product with other products. The EI teams depended on external libraries more than the Jazz team. The tag keyword most-used in the EI project was the acronym for an external library as explained by EI-L: *"That is a company that we were contracting to [...]. So we're using their libraries and their library is called [acronym]."*[13] Other keywords that fell into the architecture category are `third_party`, `vs.net`, and `arch`.

[13]The acronym is not identified for confidentiality reasons.

Collaboration. All tagging done in Jazz is inherently collaborative as each developer can see all tag instances applied by other developers and all resources are shared among the entire team. Most tag keywords relate to some kind of technical concern – a component, a requirement or documentation. However, some of the tag keywords are about collaboration (i.e., they are used to coordinate collaborative processes within the development team or to communicate with other developers).

There is evidence for this in the Jazz data: The keyword `no_code` (and the synonym `non_code`) was used to flag work items that only included changes to messages, images, or JavaDoc, but not source code. Shortly before releases, this keyword was used to communicate to other developers that working on that particular work item would not affect the functionality of the system as it did not touch the source code. Another example was `fixready`. This keyword was used by one developer to indicate that the fix for a particular work item was ready, but had not yet been committed due to a missing approval for a related work item.

In the Jazz project, tags were frequently used to coordinate changes between different component teams. J-A explains: *"`Adoption` means that there's a work item that's in our bucket or another bucket for which there's a change set attached that someone in another team has to adopt."*

Component. The tag field was also used to refine the work item categories that Jazz already provides. Compared to the categories, component-specific keywords can be introduced without any effort or official conventions: *"He could've made another heading for each of [the sub-categories]. But, for some reason I guess, the tagging was probably more open. I guess when you go and modify something like the spec, something like that; it feels very administrative, [whereas] this tagging is supposed to be more fluid."* (J-D3) Component-specific keywords were often used to categorize work items and their use depended largely on the presence of other categorization mechanisms. In both data sets, component related keywords accounted for the highest number of tag instances.

Cross-Cutting Requirements. Unlike component-specific tag keywords, cross-cutting keywords capture aspects of work items that cross cut the hierarchy of categories for work items as J-A explains: *"[Cross-cutting tags] are orthogonal to categories. They are – that's the beauty of tags that they are cross-cutting. It's not about grouping, when we have grouping then things can only be in one."* Cross-cutting tag keywords can be distinguished into functional requirements and non-functional requirements. Examples for non-functional requirements include keywords such as `performance`, `accessibility`, `scalability`, or `responsiveness`. On the other hand, functional requirements that cross-cut several components include `internationalization` and `errorhandling`. The use of tags for cross-cutting concerns was more frequent in the Jazz data.

Documentation. In both projects, tags were used to identify work items that were documentation related. In the Jazz data, there were keywords for `doc` and `documentation` (synonyms), and keywords that helped with the compilation of the "new and noteworthy" for each release. The keywords relating to documentation in the EI data included `id` (information delivery), `docs`, and `samples`. An example that shows how powerful the tagging mechanism is for tasks that require some metadata but that do not require a formal process is given by J-D1: *"I went through a bunch of things that were tagged with `faqable` or `faq` or something like that, so then when I was done, in order to see what I'd done, I tagged it with `included_in_faq`."*

Environment. Software products have to be adjusted to work on particular browsers or particular operating systems. In both data sets, a small number of tag keywords were used to indicate work items that were related to compatibility. In the Jazz data, these keywords included `linux` and `ziseries`. Developers in EI used the keyword `conformance` as EI-L explained: *"So `conformance` would be tasks related to supporting particular things, like say, a particular browser or a particular database vendor or a particular app. So product managers would come up with a required conformance."*

Idiosyncratic. Among the tag keywords that we classified, two keywords in the Jazz data set stood out as being idiosyncratic. They were neither related to a milestone, nor were they used to organize work items according to components or cross-cutting concerns. They were used for various reasons and were often only used by very few developers. The two idiosyncratic keywords in the Jazz data were `selfhosting` and `rfe`. `Selfhosting` was used in the beginning of the project when Jazz started to be self-hosted. When the entire project became self-hosted, the keyword became unnecessary; but at the time it was the easiest way of marking work items that related to the self-hosting aspect in particular. The other keyword is `rfe`, which was used to flag work items that were created by the support team for official customer requirements. Again, this was not something that could be expressed through other work item features, such as priority and severity, but nevertheless was important to record.

Planning. Tags were used heavily for planning purposes in both case studies. Keywords such as `beta2candidate` and `committed-sprint-8` showed whether a certain work item was a candidate to be included in an upcoming release or whether it was committed towards that release already. It is interesting to note that the number of instances per tag keyword for planning tags was much higher in the Jazz data. Since the EI teams mostly use Scrum, their "releases" were a lot more frequent – and included fewer work items.

In both case studies, one particular keyword was used to flag work items that definitely had to be included in a particular release. In Jazz, the keyword used for these work items was `mustfix`, in the EI data, it was called `mandatory`. J-M explains: *"I have to say easily, the most-used and the most useful tag has been the `mustfix` tag, right. Especially when we'll sort of – we're working on some effort and there's limited resources, limited time, and it's like, OK, do we really need to fix this thing or not, right? Regardless of all the other fields in the work item that tell you information about that defect, right. Bottom line is, do we need to fix it or not? And to be honest, this `mustfix` tag is usually set by the development manager, right, based on discussion with other people. And interestingly, sometimes you see in the work item, they'll sort of argue, well not argue, but they'll ask, they'll say does this really have to be `mustfix`, right."*

Work items were annotated with planning tag keywords extensively only during a specific period of time as they were related to a milestone in the development process and usually contained the name of this release (e.g., `beta2candidate`).

Testing/Debugging. Tags were also used to coordinate the testing process. In the Jazz data, keywords such as `svt` (System Verification Test), `tvt` (Translation Verification Test), and `fvt` (Function Verification Test) were prominent, and other keywords, such as `buildstatus`, were used to indicate how a certain bug was found; as described by J-D1: *"`Buildstatus` is flagging work items that I've created while I [was a release engineer] that have something to do with the current status of the build. So if it's broken and I'm complaining, I flag it with `buildstatus`."* Keywords such as `include_in_testplan` helped coordinate the testing process, and `review` was used for work items that contained reviewing work rather than development work.

Testing and debugging played a less prominent role in the EI tag data. The only two keywords that fit into this category from the EI project were `qc` (quality control) and `testing`.

Tooling. The developers in the EI case study switched to using Jazz in the middle of their projects and they did not start using all features of Jazz right away. Work item tags were helpful in that situation as they allowed the developers to create processes based on *ad hoc* artifacts. The keywords that we classified as tooling were almost exclusively annotations that could have been made to the work items in Jazz through different tool features. For example, there were at least 42 instances for each of the keywords `defect`, `enhancement`, and `bug`. Tag keywords were also used instead of team areas as explained by EI-L: *"[Our project] has 4 teams: team a, team b, team c, team d. And each of those, I think we've started off the practice of adding those items and tagging them. And I think that might have dropped off for some of the other teams, but the team b, it looks like they stuck with it."* `Teamb` was the fourth most-used keyword in the EI data. In contrast, tooling related tag keywords were not used by the Jazz team.

RQ3: The Role of Work Item Tags

This section discusses the findings for the third research question on the role of the tagging feature in the work practices of individuals and teams. To answer this question, data from the interviews and from the observations was analyzed. The archival data was used to further explain and confirm the findings from the interviews.

Tagging Audience: Self, Team, and Community. Although tagging in Jazz is by design a collaborative feature, as is the case in most tagging systems, it is difficult to verify if tagging was done to service collaborative or individual needs. In most web-based tagging systems, such as Flickr, collaboration is supported [9, 154]. At least one category of tag keywords attached to work items was explicitly about collaboration (e.g., `non-code` for communication and `fixready` for coordination). For other categories of tag keywords, the interviews revealed that these supported social as well as individual activities. This category of tag keywords is not as evident in systems such as Flickr because the focus in these systems is on navigation and categorization, rather than on supporting articulation work which is a crucial aspect of collaborative development.

Some developers predominantly tagged to service pressing individual needs, but also used tags to support collaboration within the team. As J-D1 says: *"I primarily create them for myself. But with the candidates, for example, that's obviously for someone else's consumption."* However, other developers see tagging as more of a team activity than an individual one, as J-D3 says: *"I don't personally tag work items for myself that much. But I know when I was doing the testing for the [component] that [J-D1] wrote and [he] had basically a tag for each command, so I would follow - I would add his tags, like his conventions, 'cause I figured it would be easier for him, right. And I'm already on the work item when I'm creating it, so - I would add those as I was creating them."* There was another instance of where a new team created a list of the tags already used in a project. Furthermore, J-A discussed how tagging was not just for the team: *"It's a bit for everybody, for the team as a whole and for people on the outside."*

Tag instances were added by feature owners, team managers, and other stakeholders, such as release managers. For example, this quote from EI-R demonstrates how an agreed upon keyword called `mandatory` was used to coordinate work across various team roles: *"We had a lot of features - and we had to determine which ones are mandatory for the release. And they went through an exercise through all the features, and for the ones that are mandatory, they actually use the tag `mandatory`. So product management, development mostly, and as well the execution team."* This same developer further went on to discuss how tags may be used to support the articulation work of breaking a task into sub-tasks and to support communication from management about high-level features: *"It's mostly the management, but as well, there is nothing that says development won't do it. They are the ones mostly who break some of the stories or work items into smaller items, and they will tag it appropriately if needed. But mostly the tags are at the high level, like features and so on, which is mostly feature owner, the scrum master, or the manager of the component."*

According to the archival data, 184 community members tagged work items in the case of Jazz in addition to the team members. However, the smaller participation by the general community members sets apart the kind of tagging done here from the tagging performed in Flickr and Delicious.

Tagging Motivation: Categorization and Organization. One key reason for the use of work item tags was categorization. As J-L1 put it: *"Mainly as a kind of categorization. [...] Tags are useful for identifying cross-cutting concerns like performance or accessibility or scalability or responsiveness, things like that, or testing."* While the Jazz interface already provides an opportunity to categorize work items (see the *Filed Against* field in Figure 3.1), tags are more flexible. The category tree can be altered, but this would change the available categories for the entire team and does not work for cross-cutting concerns, as EI-D2 noted: *"If we're organizing a team and we have a bunch of work items that will span different projects or different logical organizations, tag them all together, so that I can make one query for all those types of things. [...] I find it works well for things that don't quite fit into a nice tree hierarchy."* J-D3 also identified this disadvantage of the top-down classification: *"The problem is its very administrative-side feeling, which is fine, except it's not as flexible to just* ad hoc *make things."*

The developers interviewed recognized the benefits of tagging over a strict categorization scheme (which is available for Jazz work items but is strictly controlled). This quote from EI-D2 captured how tags were useful for capturing varying viewpoints: *"No, the category tree, you couldn't make it perfect, because my definition of perfect would be different from your definition of perfect and even then, I don't think in a tree [structure]. So, I have a hard time when browsing a large piece of information where I have to know how somebody else would categorize it in order to find it. [...] Different companies and different users and even people within this company are going to use it - and different teams - are going to use it differently. So I don't think you can build sort of a one size fits all type model for it."*

Tags were also seen as a way to organize work items. J-D1 responded: *"[I use tags] because I feel like [work items] should be organized. I feel like they're there and so I should use them. [...] I don't know if they do organize work items, but it makes me feel like I'm doing something when I associate a tag with it. In theory, I'd like to believe that tags draw work items that have a similar area together. That's my hope."* The organization achieved through tags was different from other categorization mechanisms as described by EI-D2: *"It allows me to kind of organize the work items in a way that is very free-form and flexible and so I can write queries that fit what I'm looking for, rather than the 1,700 different fields that there are and trying to figure that out. [...] The tag stuff is nice and free-form, and allows me to think the way I want to think."*

On the other hand, developers were not forced to use the tagging feature of work items. For example, EI-D3 explained: *"I didn't find it useful. It didn't give me any additional information."*

Tags in Use: Finding Work Items, Articulation Work, and Information Exchange. The main use case for tags was finding work items later as described by J-A: *"I use [tags] to categorize things basically, so I can have queries and find things. I'm afraid of losing work items if they aren't [tagged]."*

In particular the developers in charge of assigning work items to other developers used the tagging feature. In the agile processes followed by the EI team, this kind of articulation work was the job of the scrum master. This role rotated between developers. EI-D1 described the process: *"I have [...] to organize their stories, their tasks, etc, etc. Trying to find things that fall into their work category. I need to do searches on them to pull that stuff out of the backlog to propose it to the team. And [in] doing so, this is the type of queries that I would use the tags for. [...] A lot of times you can get that information through queries of just the title or the description, but I also like to do the tags, 'cause if somebody actually did use use them, then it'd be good. [...] So I don't go searching for those things. I don't go out trying to specifically enter these things I should say. But when I'm creating queries and such like that, I do look to see what's available and I will use that."* In this case, tags helped for exploring the work items: *"They have led me on paths to find – or think about other things to search for. So, for example, if I do a [...] search on it, and a [certain tag appears, then I] read up on that particular item, figure out what it's involved in, and then from that, I can do other queries that find items in my component that will base on that."* Similarly, the release engineer EI-R used tags as well: *"I rely completely on using these [accessibility] tags [...] to pick up on these requests. [..] And we know when the request comes, through the queries."*

In the Jazz project, the administrator J-A searched for tag keywords: *"For non-functional things, like usability – Okay, we got two weeks to do some things, what are the usability related enhancements that we have, for example. So we have a set of work items, that are enhancements, and some are usability related. So I look at those, I go 'Oh, these are easy. Let's try and fix these four.' For example, during the polish phase of the 1.0 release, we had two weeks to polish. [...] So we had tags like* ***`polish`*** *and* ***`usability`*** *and I use that to kind of guide what work items we could work on."*

But tags were also used for queries by developers who were not in charge of assigning work items. For example, J-L1 described: *"I used them the other day, trying to search for [...] a list of bugs against [a related product]. Thinking that, you know, I was probably a good boy and had tagged any [product] related issues with the [product] tag and did a search for [product] tags and that actually found very few. [Laughs] 'Cause I hadn't been a very good boy and tagging my Jazz work items with the [product] tag."*

A specific case of tag use in queries are the dashboards in Jazz that are displayed and configured using the web interface. Dashboards are intended to provide information at a glance and to allow easy navigation to more complete information. By default, each project and each team within a Jazz project have their own dashboard; and an individual dashboard is created for each developer when they first open their web interface. A dashboard consists of several viewlets. Viewlets are rectangular widgets displaying information about an aspect of a project. Developers can add viewlets to their dashboards and configure the viewlets using different parameters (see Section 4.1). J-A reported: *"We have a lot of dashboards that are tag-based, like the test teams, favourite bugs and so on. Having a tag lets us have those [dashboard] viewlets that otherwise – it would be really hard to describe a query that says, 'Show me all the work items that were added by the test team'."*

J-M introduced a particular keyword to ensure visibility in the project's dashboard: *"There's things that we specifically track. I introduced this* ***`tracking`*** *tag, so there's a* ***`tracking`*** *tag that we put on certain kinds of work items which actually then show up in a dashboard. [...] There's something where we just want to raise the visibility."* As J-A described, tags increased awareness: *"On my team anyway, if I think it's related to one of those characteristics like performance, I tag it because I want to be aware of it and I want to have a query that shows me what they are."*

Many of the tag categories discussed previously (such as component and cross-cutting requirements) were to add information to work items. These tag keywords also played an awareness role in informing others and keeping them informed about work items: *"My use for tags mostly is just to get an idea of what other work items are about. [...] I think the PMC [Project Management Committee] likes to put like* ***`1.1candidate`*** *or* ***`0.6candidate`****, so like that - that rarely impacts me, but I see it and then I'm just a little bit more aware, so that okay, sure, someone finds that important."* (J-D3)

Tag Management: Keeping Track of Tags, Removing Tags, and Tag Structures. Teams of developers developed mechanisms or informal processes to help manage the tags they used on a project. For example, EI-D2 discussed how he externalized lists of tag keywords to help in maintaining consistency: *"I have been keeping a list of tags so that I don't, you know, create duplicates, things that are spelled alike or that sound alike, or two different ways of referring to the same thing."* Externalizing lists of tag keywords was used as a mechanism by a new team to learn which tags were used in a project: *"There was a new team that joined and they were like, what tags should we use? We don't actually have them written down anywhere. So they went and they did compile a list – I don't know where they put it, they put it somewhere on the wiki – right, of the tags. Well, they asked me and I said, here's [...] off the top of my head [...] about 5 or 6 that we use a lot, right."* (J-M)

Developers respected their colleagues' tags and were reluctant to remove tags: *"I don't believe I ever have [removed tags] and I'd probably be reluctant to. And it would - if I were to – it would probably have to be something that I own or am deeply involved with, right. Because, again, if my presumption is, if they're more like a supplementary thing, if a tag is wrong, I'm not going to actively go out and say, no that's wrong and fix it, I'm going to let whoever owns it make that call."* (J-D3) Developers considered owners of work items should manage and remove tag instances on those work items: *"I would only [remove tags] if I attached the tag to the work item or if I owned the work item. I wouldn't want to mess with - I don't know - someone else's categorization."* (J-D1)

Tag instances were not removed often, however, since work items do not show up in queries anymore by default once they have been closed. As EI-R described: *"We use [the tags], then we just forget about them."* EI-D2 explained: *"Once I tag something, usually what happens is I close the work item and it'll remain tagged because it's still part of whatever that is. [...] I mean unless I made a mistake or mistyped something, I can't think of a time where I actually went in and said, 'Oh this a stupid tag' and removed a pile."*

There were only two scenarios that were mentioned by interviewees where tag instances would get removed from work items. The first one was about the status of the work item as described by J-A: *"I'm tagging these things for candidates, [and then I find out] they're not [candidates] and I untag them all."* The other example was given by J-L1: *"If I don't like the way somebody tagged it or there's a better tag for one that they used, I'd remove it and add another one. But that doesn't actually* *happen terribly often."* This issue of removing tag instances is not something that occurs in Flickr or Delicious because the tag instances are not directly attached to the resources as they are in Jazz.

J-M mentioned the need to manage the complexity of tag structures that were emerging in the tagging vocabulary. Some component tag keywords as well as some testing keywords had refinement keywords associated with them (e.g., `testing.performance`). J-M discussed both of these structures and specified a need to monitor how that structure was being decided about in email discussions: *"I wasn't involved in the discussion, I was watching the email about it and then they decided that, no they were just going to use the tags instead, and then listed all the 10 different tags and so in fact I was thinking about that this morning, OK, I need to get in on that discussion, saying, no, I don't really like it that way."*

Vocabulary Consensus: Explicit, Implicit, and Tolerance. An important aspect in understanding the advantages and disadvantages of tagging systems, especially in terms of representing knowledge and task coordination, is if the community using the system will converge on a common vocabulary.

Sometimes consensus was reached in an explicit way (e.g., through emails, meetings, and wikis), and an explicit consensus was particularly important for the planning-related tag keywords. An example of this was shown above in terms of the `mandatory` tag keyword. Several more cases on how planning-related tag keywords were agreed on emerged in the interviews. Two examples included: *"I usually tag with a tag whose name I've been told. [...] Or an email that's sent out. So,* ***`m1candidate`****, for example."* (J-D1) *"I already know this - he had set a convention, so I was following the convention. Other than that, the other tags, I kinda see or pay attention to, is during milestone releases or candidate releases."* (J-D3) Component keywords were explicitly agreed on as well, for example as explained by J-L1: *"Either myself or the team lead would establish a tag for the area, so - in the [component] team I established one called* ***`workspaceeditor`*** *and, you know, I would use that consistently and other people would start using it as well once they saw it."*

The list of available tag keywords was externalized either in a wiki or a list to encourage their use. But at the same time, developers were not insisting that the tags were used as described by EI-D2: *"Well, I guess that I keep the central list of - and that list is in a publicly available area. So I assume other people do look at it and I've had a few questions on it so I think other people look at it but I can't tell you* *whether everyone does and everyone follows it - and frankly I wouldn't want it to be that hard and fast a rule. It's supposed to be kind of a loose system, so I keep that for my own convenience."*

Although some keywords were agreed on in an explicit way, other tag keywords were frequently agreed on through an implicit means. This interview quote from EI-R demonstrates how explicit and implicit mechanisms for consensus occurred: *"Maybe the only one that at least I know of [that] has a convention is the one that I put because I make it tight to the integration build that we are in, so we know where the approvals went. But other than that, if it's legal or other than that, there's not really conventions - just a word that means something, and whoever is basically working on these work items, knows what it means."*

Implicit awareness of tag options can occur through watching work item feeds: *"You just sort of see it happening in the work item's feeds"* (J-L1), or through content assist: *"There's always the concern about when you're creating tags, are people going to create more tags? I mean at least the good thing right now is that when you create a tag it tells you that you're creating a new tag."* (J-M). Consensus can be achieved through the context and existing knowledge underlying the use of the tag keywords: *"If you're part of the organization - it would be odd that somebody would use a tag within the organization that means nothing to somebody in the organization. [...] All of the tags that I have seen that are there, I'm aware of what the acronym means, what the word means, in my context"* (EI-D1).

To further facilitate implicit understanding, the developers took steps to make sure that the tag keywords they created were comprehensible to other team members: *"I make sure I don't use too many acronyms, so that people can understand what the tag means. Apart from that, I try to make it somewhat descriptive, so [...] I wouldn't use `perf`, instead use `performance`"* (J-A). There was trust that this process worked: *"Anything that makes sense, intuitively to me, I just go ahead and tag it. Hopefully what makes sense intuitively to me will make sense to other people as well"* (EI-D2).

This last quote captures that there was not a large concern if tag keywords were misunderstood, and that there was tolerance for some variation in keywords. This is important because there may be concern that ambiguity of tag keywords and the use of synonyms may lead to problems in using tagging systems [147] but the interviews did not reveal such issues: *"And then, after having seen something for so many times – not that I would necessarily know if there's a difference between, you know, slight variations in tag naming, but if I've got an idea, I'll do it. And then again, I wouldn't* *be too afraid if I got it wrong, 'cause someone will just change it, right"* (J-D3).

Table 3.4: Most frequently shared tag keywords in Jazz

tag keyword	#instances	#distinct users
`performance`	413	46
`globalization`	442	45
`tvt`	636	45
`polish`	966	43
`maintenancecandidate`	436	40
`no_code`	197	40
`errorhandling`	431	38
`usability`	441	38
`beta2candidate`	308	35
`rc4candidate`	133	33
`ux`	668	33

Table 3.5: Most frequently shared tag keywords in EI

tag keyword	#instances	#distinct users
`id`	481	30
`a11y`	433	24
`docs`	427	24
`mandatory`	185	21
`conformance`	336	20
`defect`	132	17
product X	113	15
`third_party`	300	15
product Y	79	14
`all`	205	13
`teamb`	466	13

Tables 3.4 and 3.5 show the most shared tag keywords for both projects.

Bridging Lightweight and Heavyweight Task Organization. Task management systems have traditionally focused on a limited set of categories per work item, such as work item summary and description, a unique ID, the creator and the owner of the work item as well as priority and severity. As modern IDEs add explicit support for collaboration, fields such as comments and tags have been added to the set of common categories of a work item. While there has been some research on how to improve the quality of work items from the perspective of a bug reporter (e.g., by Bettenburg *et al.* [18]), there is surprisingly little work on task management. Categories of work items, such as priority and severity, are often taken for granted and rarely questioned. A notable exception is a study by Herraiz *et al.* [88]. Based on the observation that there are fewer clusters of work items than severity levels, the authors conclude that there is a need to simplify the category set of a work item.

Work item categories, such as priority and status, can be seen as heavyweight because there is a limited and formally defined set of values that can be assigned to each of them. Changing this set of values is often impossible as it affects the entire project and would require updating all work items in a system. On the other hand, with the shift towards built-in support for collaboration, mechanisms such as tagging and commenting have been added as an additional way to categorize work items.

Based on the data collected from IBM's Jazz project, I examined this phenomenon in more detail. Shortly after the 2.0 release of the Jazz client RTC in June 2009, three categories were added to the task management system. Table 3.6 shows the category names as well as the values that could be selected for each of the categories. The tag data was analyzed to find out which of the values that could be selected for the new categories had pre-existed as tags. For most of the values, a corresponding tag existed. These tags are noted in Table 3.6. *How Found* contains information about how a certain problem in the source code was found (e.g., through testing or through customer use), *OS* identifies the corresponding operating system, and *Client* names the platform and version that the task belongs to.

The assumption that the new categories were added because of the tags was confirmed through the interviews: *"Recently we added some new fields on the work item. For the defect, we added a client and so forth. So we're starting to replace some of the tags. Deprecate some of the tags by having fields for them in most cases."* (J-M) This phenomenon was also described as "training wheels": *"That seems to be almost training wheels – or it seems to be a way of trying out a field for work items. [...] Defects have different fields, platform and other things. And we used to use tags for that, but now it's been promoted. That seems to be what tags really get used for, a cheap way to add fields to work items. I'm not saying that it's bad. But I'm saying that that seems to be how it works out."* (J-D1)

One of the issues that arises from the adoption of new work item categories based on tags is a large number of categories that might become overwhelming. As one of the interviewees pointed out, adding categories that are not always necessary can

Table 3.6: New categories with possible values, and corresponding tags

category	allowed possible values	tags
How Found	Unknown, Customer Use, Self Host, Development, Test, Internal	testing, selfhosting, selfhost, self-host
OS	Unknown, Windows, Linux, Mac OS, AIX, Z/OS, i5/OS, Other	windows, linux, mac, aix, ziseries
Client	Unknown, Eclipse client, Visual Studio client, Firefox 3.0, Firefox 3.5, Internet Explorer 7, Internet Explorer 8, Safari 3, Safari 4, All browsers, Command-line interface, Other	eclipse, firefox, ie, safari, browser

have disadvantages: *"We're actually at that point right now. There's a client field, I think. There's something that lists how the problem was found. So it lists five web browsers, the CLI, and the Eclipse client. And really, that field should only apply to work items that deal with the web UI. 'Cause it was added for the web UI."* (J-D1) A new category would never be reversed once added: *"I don't know how easy it would be to remove attributes from work items, though. I think that we'll never lose attributes, we'll only gain them."* (J-D1)

3.1.5 Discussion

In this section, recommendations for how tool support for tagging could be broadened for other social and technical artifacts are discussed, as well as how such tool support can be improved.

While tags have already been adopted by the software developers in this study, there are still areas where tool support for tagging can be improved. However, the eagerness with which tags have been adopted and the experiences of the interviewees suggest that the informal nature of tags ought to remain intact. EI-D2 explained: *"Part of the beauty of it, I think, is that you only have to enter, you know, a word or two, and that's it. [...] I think keeping it light is good. It's actually probably the best about it, frankly. [...] The minute you start making me tag stuff, I'm going to resent it. [...] I find that the tags are a good way to reduce the amount of time that I'm looking for particular things – then it's worth the investment, but the minute I have to do something, then it just becomes a drag."*

Therefore, enhancements of tool support must recognize the current benefits of tags and the main theme of any changes to tool support should be to help developers use tags. The following tool enhancements are suggested:

- Using the same informal approach as with tags for work items, a tag property could be added to other kinds of artifacts, especially source files, test cases, and requirement documents. Tags can also be implemented on a fine-grained level (e.g., for methods and fields). This would enable tagging across different types of content and thus would further support collaborative organization of artifacts. Similar ideas have been successfully tried and tested in TagSEA [160]. Tagging for builds has recently been added to Jazz, but the tagging systems are treated entirely separately. It is not possible to retrieve all work items and builds tagged with the same keyword through a single query.

- Display tag authors along with the tag instances. During the interviews, I showed the interviewees a list of tag keywords that were used on their work items but that they did not apply themselves. The participants used the tag authors to understand the keywords. Adding the author information of tag instances is not obtrusive as the information is collected anyway and could just be displayed on mouse-over.

- Apart from author information, the only meta data property that should be added to tags is an optional description. For tag keywords that do not have an obvious meaning, such as `adoption` or `buildstatus`, a short description would increase the usefulness as there are tag keywords in the vocabulary that may be unfamiliar to some developers. When a new keyword is introduced to the vocabulary, a dialog could ask for an optional description instead of just notifying developers that they are about to introduce a new keyword.

- Current tools do not offer any management for tags on work items. Useful functions would be changing all tag instances with a particular keyword (e.g., to fix spelling mistakes). For synonymous keywords, such as `doc` and `documentation`, folding would be beneficial. Similar refactoring mechanisms have been implemented in TagSEA [160].

- To increase understanding of how tags are used and which tags are suitable for work item search, a way of externalizing tags should be added. Information about tag keywords and instances, their authors, the corresponding work items, and the time of the tag instance creation is available in the system, but this information is not used yet by the work item tooling. The explicit mechanisms and externalization activity for reaching tag consensus among all users are not activities that are likely to occur with users of Flickr or Delicious. In the case of Flickr, Ames and Naaman report that two of their participants coordinated tags for photos with others in order to facilitate later search and retrieval [9]. However, the consensus described here in the context of Jazz goes beyond that and includes all members of a project.

- Once the tag vocabulary is analyzed, tags for incoming work items could be suggested. Strong candidates for suggestions are tag keywords that have extensively been used in the near past, such as planning-related keywords and keywords that have been applied to work items in the same category.

When tags were initially introduced in Jazz, several additional features were suggested by developers. However, over the time of more than three years of tagging activity, developers adapted to the tagging tool support as it was initially implemented, J-A said: *"In the beginning, I thought that we should do a lot, have private tags, have more meta data with them. In the hindsight, their simplicity is kind of interesting."* Therefore, it is important to not introduce barriers, and to focus tool enhancements on meta data that can be automatically collected, such as author names.

3.1.6 Limitations

When the Jazz team started on their project, the tagging feature for work items had not been introduced yet. This might have influenced the specific tagging behaviour. Also, the Jazz developers might be biased towards their own tool and their usage pattern might be different from other developers. However, the findings were confirmed with other development teams that are much larger than the Jazz team. There might still be bias as all development teams in the study were part of IBM. However, the team members of the EI case study are not related to the Jazz project and work in a completely different domain.

In our analysis, we only analyzed the "heads" of the "long tails" in the tag distribution that account for 80% of all tag instances. Although the remaining tag keywords may not have been used frequently, there may have been very important categories that should be examined. However, a fine-grained analysis of all keywords is beyond the scope of this research. An alternative could have been to code a random sample of the tag keywords. We decided to focus on the most frequently used ones, but considered all tag keywords mentioned during the interviews.

Our interpretation of the tag categories could have potential errors. This issue was addressed by asking about specific tag keywords and instances in the interviews, by reading the summaries and descriptions of the corresponding work items, through follow-up emails to the participants, and by searching the project websites and pertinent mailing lists. Potential errors are also offset by the ethnographic-style observations that were conducted on the Jazz site for seven months and at the EI site for two weeks.

3.1.7 Summary

The studies described here have shown how the social computing mechanism of tagging has been adopted and adapted by two large software development teams. Not only is tagging used to support informal processes within the teams, it has also been adapted to the specific needs of software developers. Different kinds of tags have emerged over the duration of a software project for processes that require meta data but are not formalized, ranging from architecture and planning to collaboration and testing. The main advantages of using tags in software development are their flexibility and their informal, bottom-up nature. While fields such as *Milestone* or *Cross-Cutting Concern* could be part of fixed schemata, this would add overhead for work item creators and owners. Tags add the same functionality without implying administrative changes.

This section introduces the ConcernLines tool that was developed based on this research. ConcernLines visualizes the use of tags over time, and thus allows developers or managers to gain additional insights into the tagging activity.

3.2 ConcernLines

The previous section described how developers document relevant concerns using tags on work items. To facilitate the exploration of how developers use tags, the ConcernLines tool was created. The tool was used to create timelines of how tags emerged and co-occurred over time. When the tool was shown to the participants of the studies described in the previous section, their feedback indicated that a tool for viewing information on co-occurring tags could be very useful. For example, developers wanted to answer questions such as: Which components played a key role during the last beta release? Which non-functional requirements co-occurred with work on the user interface? Which components were affected by the improvements on user experience? Their reaction to the tool and their suggestions for enhancements prompted further tool extensions so that developers could use ConcernLines during development to understand the emergence of co-occurring tags over time.

ConcernLines visualizes the evolution and relevance of tags over the lifetime of a software project and enables the identification of co-occurring tags. It displays the evolution in a snapshot, using horizontal timelines for individual tags and mapping the relevance of tags (e.g., determined through frequency of tag occurrences) onto colour intensities. Co-occurrences can be identified by pivoting on one tag in a selected time range.

ConcernLines can be used to visualize data other than tags, in particular all data that can be seen as a concern at a given time. Other than tags, there are various data sources that can be mined to determine concerns (e.g., source code, source code comments, JavaDoc, CVS commits, work items, or email communication). According to IEEE, concerns *"are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders"* [97]. There are different dimensions of potentially overlapping concerns [131], ranging from process-related concerns, such as milestone releases, to requirements, such as usability. Determining relevant concerns as the system evolves is not always easy.

The term *concern* is often used synonymously with the term *aspect* in the context of aspect-oriented programming (AOP) [100]. In contrast to that, the definition used here is broader in the sense that it goes beyond source code. Aside from AOP, important tools for concerns in software development include Concern Graphs [144] and ConcernMapper [145]. They enable developers to associate parts of source code with high-level concerns. Opposite to this top-down approach, the tool TagSEA [160] takes a bottom-up approach of relating source code to concerns by allowing developers to tag their source code. Mapping of concerns is also supported by Hyper/J [131], a tool that focuses on the separation of concerns in multiple dimensions and thus acknowledges the existence of different kinds of concerns. Here, the focus is on concerns determined through work item tags.

The remainder of this section is structured as follows. In Section 3.2.1, related work on visualizations, focusing on software evolution, is summarized. ConcernLines is introduced in detail in Section 3.2.2 and three example scenarios for the tool are presented in Section 3.2.3. Section 3.2.4 summarizes this discussion of ConcernLines.

3.2.1 Related Work: Visualization of Software Evolution

Ultimately, the visualization of concerns over time can support our understanding of the evolution of a software process. Understanding the evolution of a software development process is a prerequisite for efficient management, improving processes, and the integration of members into development teams. Gaining high-level evolutionary information about large software systems is a key challenge in dealing with increasing complexity and decreasing software quality [70]. While several tools for evolutionary information at the source code level have been proposed, research on the visualization of process information over time is limited. However, process understanding has been identified as an important aspect of software maintenance [190].

Literature on visualizations of software evolution can be categorized by the data that is represented. The main categories are source code, commit events, metrics, and data derived from work items.

On a fine-grained level, Voinea *et al.* looked at the evolution of source code on a line-by-line basis with their tools CVSscan [190] and CVSgrab [189]. Eick *et al.* developed Seesoft [56], a software visualization that maps each line of source code to a thin row and uses colours to indicate changes.

A tool for the visualization of commit events was proposed by Telea and Voinea [167]. Their Project evolution view maps different files to the y-axis and time to the x-axis. The matrix is then filled using different colours for commit events. Using a visualization like this, clusters of files that are changed together can be identified. Another timeline view of source code files focusing on cluster recognition was proposed by van Rysselberghe and Demeyer [151].

A first step to aggregate information at a higher level of detail and displaying its evolution was done using metrics. The Evolution Matrix by Lanza and Ducasse combines metrics with evolution visualization [106] and the Evolution Spectrographs by Wu *et al.* provide a metric-based representation of a software system's development history [192].

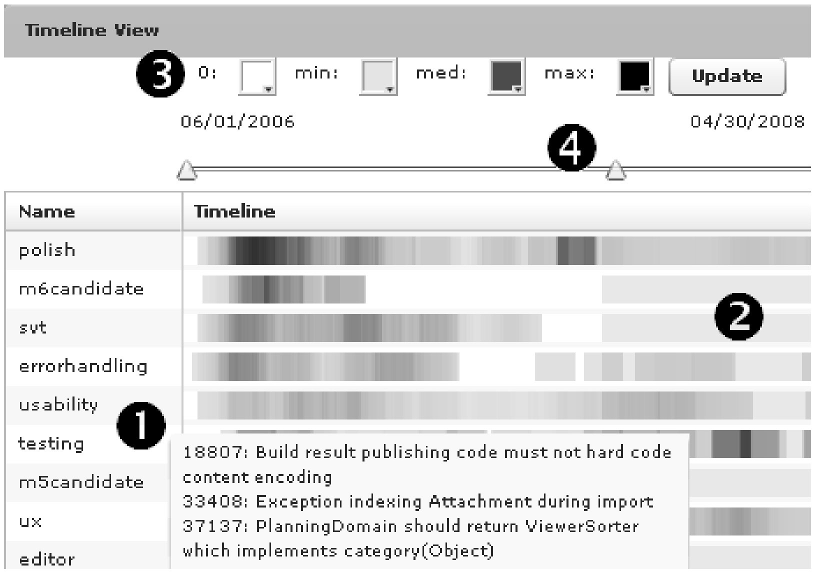
Research on the visualization of data derived from work items is still limited. The Discrete Time Figure by D'Ambros and Lanza [43] is a visualization technique in which historical and structural data are embedded into one figure. Software entities are shown as horizontal timelines, with the number of commits regarding these entities and the number of bugs reported mapped onto the timelines. Also leveraging the information available from work items are Deep Intellisense [92] as well as a tool proposed by Fischer and Gall [61]. They focus on integrating historical information from various sources and the evolution of dependencies between features.

In addition to the static tools discussed above, animations that use video or interactive tooling to display evolution have been proposed. Prominent examples include Evolution Storyboards [19] for visualizing a series of software graphs and Gevol [36] for visualizing CVS data.

3.2.2 The ConcernLines Tool

ConcernLines aims to visualize concerns in software evolution over time and to allow the identification of temporal concern co-occurrences. Like most of the tools discussed in Section 3.2.1, it utilizes a timeline view to represent time. Before explaining the details of the implementation, the visualization is defined using the five dimensions proposed by Maletic *et al.* [119]: task, audience, target, presentation, and medium. The main task supported by ConcernLines is the exploration of characteristics and interrelations of release history, non-functional requirements, and software components based on concerns. The intended audience reaches from management for high-level insights to new team members for familiarization with a given project. The data source represented (i.e., the target) is a mapping of concerns extracted from software artifacts (here: work item tags) to time, and the representation is a timeline-oriented view of concerns over time using a regular display as the medium.

Interface



18807: Build result publishing code must not hard code content encoding
33408: Exception indexing Attachment during import
37137: PlanningDomain should return ViewerSorter which implements category(Object)

Figure 3.7: ConcernLines user interface

Figure 3.7 shows the user interface of ConcernLines. The main components are the timelines (2) for the concerns listed on the left-hand side (1). In the current implementation, this part is enhanced by a tooltip with relevant work items. The interactive part of the tool consists of support for customizing the colour scheme (3) and a time range selector (4). Unselected time ranges are greyed out in the timeline part (2). ConcernLines was developed using Adobe's Flex[14] and is accessible through web browsers. The tool uses a CSV file with a matrix of concerns and their intensity over time as input data. Such a matrix can be computed by mining repositories of software artifacts, such as work item tags.

Timelines

A timeline represents each concern (for the examples here, tags on work items are used as concerns). The timelines are rendered from left to right according to time.

[14]http://www.adobe.com/products/flex/

As observed by D'Ambros and Lanza [43], timeline visualizations do not necessarily scale well. To ensure the scalability of the visualization up to long periods of time, the width of each time unit is calculated dynamically (i.e., a longer time range results in a narrower display of time units). In addition, the scalability is improved by a default setting of using 30 day averages instead of the actual values in the display. Thus, two time units next to each other are less likely to have values that differ significantly.

Colour intensity is used to represent the relevance of a concern at a particular point in time. This relevance is given by the number of times a concern occurs on a particular day. Colours are user-configurable. With the default settings, white is used to represent that the concern is not relevant at a given point in time and stronger colours represent higher incidence. The suggestion to distinguish more relevant concerns was made by one of the developers viewing the first iteration of ConcernLines.

Time Range Selection

Using a horizontal slider with two end controls, the time range can be narrowed down to a range of interest. Unselected parts are greyed out in the visualization. The time range selection is used for detecting co-occurrences of concerns in the specified time frame.

Detection of Co-occurrences

The detection of co-occurrences is done by pivoting on one concern in the selected time range. The feature of displaying co-occurring concerns was suggested by one of the developers viewing the earliest version of ConcernLines: *"Api or ui or ux, [...] you might see these line up with these, with the breaks."* (J-L1) Clicking on either the concern name or its timeline, this concern is moved to the top of the list. All other concerns are arranged below it, ordered by the similarity of their timeline to the timeline of the pivot concern in the selected time range. The similarity is calculated by iterating over the days in the selected time frame and summing up the squared differences between the corresponding values (i.e., a difference d between two timelines a and b is defined as $d = \sum_{i=m}^{n}(|a_i - b_i|)^2$ for a time frame from m to n).

3.2.3 Example Scenarios

This section gives three exemplary scenarios for situations in which ConcernLines can be useful. The example data stems from the study described in the previous section.

Understanding of Tag Usage over Time

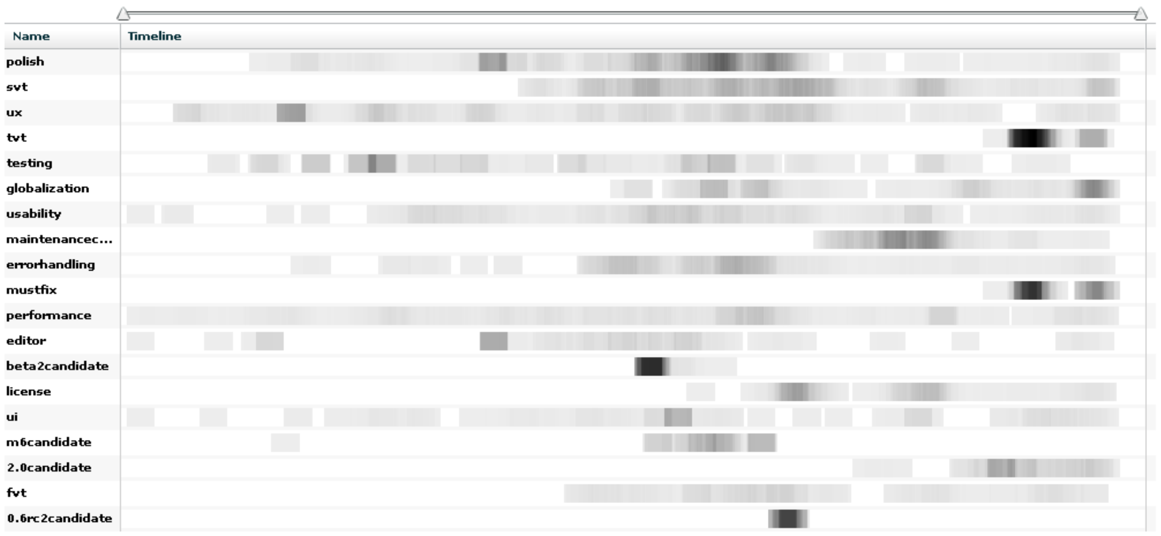


Figure 3.8: Screenshot of ConcernLines

The screenshot in Figure 3.8 shows the timelines of the most-used tag keywords in the Jazz project. Colour is used to show the intensity of tag use on a particular day[15]. Planning tag keywords, such as `beta2candidate`, have a relatively short timeline, whereas other keywords, such as `polish` and `ux` (user experience), have been used throughout the entire project. Compared to the other kinds of keywords, planning-related tags are transient, thus confirming the findings with regard to the different kinds of tags from Section 3.1.

Non-functional Requirements during Milestone 6

To get a high-level overview of what milestone 6 was about, a project manager can use ConcernLines and select `m6candidate` as the pivot concern. All other concerns will be ordered by their timeline-similarity to the pivot concern, as shown in Figure 3.9. `M5candidate` and `m6candidate` have a big overlap, and `globalization` was the main concern during those milestone releases. This view reveals the major theme without requiring the project manager to look at individual work items.

[15]Colours are shown as shades of grey here.

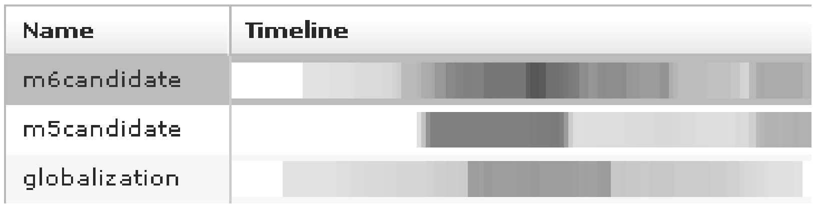


Figure 3.9: Concerns during milestone

Concerns regarding UI

To understand the importance of different cross-cutting concerns, such as usability or performance, regarding the user interface, a software developer can choose the corresponding concern as the pivot element. Figure 3.10 shows the result. In addition, the time range has been narrowed down to a time frame in which the concern `ui` has high intensity. Several conclusions can be drawn from the visualization: the concern with the highest similarity to `ui` is `svt` (i.e., testing). `Ui` also has a high correlation with `usability` and mainly co-occurs with milestone 6. The developer can conclude that the concern of usability is highly related to work on the user interface and may put a stronger focus on usability during design decisions.

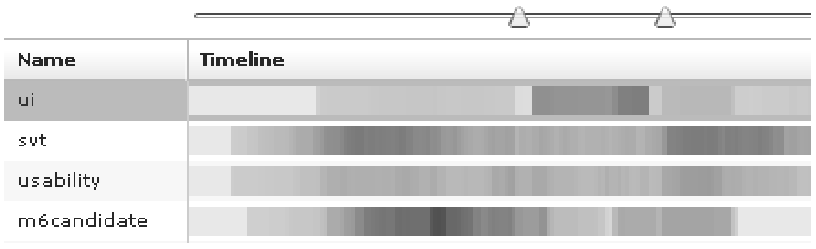


Figure 3.10: Concerns regarding UI

Limitations

ConcernLines enables developers and managers to reason over concern data, such as tags. By its nature, this data can be imprecise, and there are risks involved when reasoning over imprecise data. For example, there are no guarantees that all work items that are relevant to a particular release are tagged with the appropriate keyword, and different developers might have different motivations for using certain tags. However, these risks are inherent in the underlying data, and they are not the result of representing the data through ConcernLines.

3.2.4 Summary

ConcernLines is a tool that visualizes the emergence and co-occurrence of concerns over time. Horizontal timelines are used to represent concerns and the intensities of concerns over time are mapped onto colour. Pivoting on one concern in a specified time frame allows the user to arrange concerns by their timeline-similarity to a pivot concern. This reduces the complexity of identifying co-occurring concerns.

Concerns can be derived from multiple data sources and it will depend on the situation at hand which attributes or characteristics should be interpreted as concerns. Potential sources for concerns include keywords from work items, JavaDoc as well as other annotations from source code, and themes extracted from e-mail communication. The feedback received from the participants in the tagging studies suggests that the example used in this section, tags for work items, is a valuable choice and allows useful insights into the evolution of a software project.

Chapter 4

Awareness using IBM's Jazz

The previous chapter described how software developers using IBM's Jazz annotate work items with tags as part of their task management. However, only anecdotal evidence was shown to explain how software developers *use* the different properties of tasks after they have been configured. This chapter explores how the combination of highly configurable project, team and contributor dashboards along with individual event feeds play a role in task management.

First, Section 4.1 describes the role of two further social media mechanisms in task management: dashboards and feeds. Through a detailed study with one development team and additional data from another four teams, I examined the role of dashboards and feeds in supporting development activities, and I investigated the advantages and disadvantages associated with these tools as well as the impact of their combination on collaborative development.

The main contribution of this study is the identification of the different ways in which dashboards and feeds support task management in software development and how the interplay of these tools provides awareness on different levels. In particular, I present how dashboards become pivotal to task prioritization in critical project phases and how they stir competition while feeds are used for short term planning. These findings indicate that the distinction between high-level and low-level awareness is often unclear and that integrated tooling could improve development practices.

Based on the results, we developed a tool called WorkItemExplorer (Section 4.2). WorkItemExplorer is an interactive environment to visually explore task data. WorkItemExplorer enables developers and managers to investigate trends and correlations in their task data by making exploration flexible and interactive, and by utilizing multiple coordinated views. Our evaluation of WorkItemExplorer shows that the tool is able to answer questions developers ask, while enabling them to gain new insights through the free exploration of data.

4.1 Dashboards and Feeds in IBM's Jazz

The results on the use of dashboards and feeds stem from an empirical study of several large development teams, with a detailed study of IBM's Jazz team and additional data from another four project teams that are part of the IBM's Enterprise Infrastructure (EI) project[1]. I examine the role of dashboards and feeds in supporting development activities, and I investigate the advantages and disadvantages associated with these tools as well as their impact on collaborative development. While many research tools aiming to provide awareness of some sort have been implemented, only very few have been evaluated in an industry setting, and there is no comprehensive theory on awareness needs and the interplay of awareness tools in software engineering. Studying the early adoption of dashboards and feeds in Jazz presents a unique opportunity to begin contributing to knowledge on how an IDE should provide a comprehensive mechanism for awareness.

To gain an understanding of the role that dashboards and feeds play in awareness, data was gathered through the inspection of project repositories, by conducting interviews with developers and project managers, and through a web-based survey. The main contribution in this section is the identification of the different ways in which dashboards and feeds as provided by Jazz support task management in software development and how the interplay of these tools provides awareness on different levels. In particular, I present how dashboards become pivotal to task prioritization in critical project phases and how they stir competition while feeds are used for short term planning. These findings indicate that the distinction between high-level and low-level awareness is often unclear and that integrated tooling could improve development practices. Tool enhancements based on the results of this study are also suggested.

The remainder of this section is structured as follows. In Section 4.1.1, related work on awareness in software development is reviewed before introducing the awareness features of IBM's Jazz in detail (Section 4.1.2). The methodology is described in Section 4.1.3. Section 4.1.4 presents the findings on the interplay of dashboards

[1]The project name is obfuscated for confidentiality reasons.

and feeds to provide high-level as well as low-level awareness. The findings and limitations are discussed in Sections 4.1.5 and 4.1.6 before summarizing this research in Section 4.1.7. Following this, the WorkItemExplorer tool that was developed based on the findings from this work is presented in Section 4.2.

4.1.1 Related Work: Awareness in Software Development

The success of software projects largely depends on the effectiveness of communication and coordination within teams [103]. As the complexity of software systems increases, maintaining awareness of the overall status of a project and gaining an understanding of current bottlenecks becomes a challenge. Awareness tools that display information solely based on source code become insufficient as individual developers need to take on managerial tasks. Rather than source code alone, the status of the project arises from an aggregation of data on open and closed development tasks, successful and failed builds, delivered and pending changes, and successful and failed tests as well as evolutionary information.

In recent years, development environments are becoming more advanced with respect to awareness support. In particular, IBM's Jazz software development environment includes feeds and dashboards that aggregate data to improve awareness of high-level and low-level aspects. However, with these advances comes the need to further our understanding of what the most appropriate toolset is [104]. Most of the related work on awareness tools for software development has focused on low-level code-specific tasks rather than higher levels of abstraction [155]. There is still a lack of understanding on how to achieve high-level awareness regarding project management concerns with low-level awareness of more fine grained activities, such as source code changes and development task creation.

Maintaining awareness of each other's activities, the overall project status and current tasks is challenging as it is not limited to simple metrics based on source code or the fulfilment of a single plan item. Dourish and Bellotti define awareness as *"an understanding of the activities of others, which provides a context for your own activity"* [52]. In complex software development processes, awareness includes being aware of technical and social aspects of the development [46] as well as current and upcoming articulation work [125]. Depending on the context of the task at hand, the required granularity of this awareness can vary significantly.

Many researchers have recognized awareness as an essential part of collaborative software development and collaborative work in general. In their studies on groupware, Gutwin *et al.* found a need for workspace awareness to sustain effective team cognition, and a need for detailed awareness of one another [76]. Awareness can be maintained in an active or in a passive manner. While active mechanisms require the explicit generation of awareness data, such as directed messages, passive awareness exploits information already available in a shared workspace [52]. Even if achieved passively, maintaining awareness of other's activities and the project status is time-consuming [138] but it is indispensable in the construction of mental models of a software project [108].

Tool implementations that support awareness have traditionally focused on source code and information available from source code management systems. An early example is Seesoft [56], a software visualization that maps each line of source code to a thin row and uses colours to indicate changes. Augur [67] adds software development activities to a Seesoft style visualization, thus allowing for the exploration of relationships between artifacts and activities. FastDASH [21] uses a representation of a shared code base to highlight current activities aggregated at the level of files and methods. The workspace awareness tool Palantír [152] follows a similar approach by providing insight into workspaces of other developers, in particular artifact changes.

These tools allow for insights regarding the current activities in a project. However, in large projects, they often fail to provide an overview of the overall status of a project. Researchers have recognized this challenge and have started to develop tools that allow for higher level insights. Social Health Overview [57] works at the level of work items and mines the history of development artifacts to reveal social and historical patterns in the development process. CruiseControl[2] achieves high-level insights through the use of a dashboard that focuses on builds rather than source code. The World View in Palantír also addresses *"awareness in the large"* [153]. It provides a comprehensive view of the team dynamics in a project, particularly the geographical location of developers.

Most of these tools lack evaluation in industry settings. A notable exception is WIPDash [96], a large screen visualization for small co-located teams designed to increase awareness of tasks and source code activity. WIPDash was evaluated with two small teams during a one-week field study. In contrast, the teams in this study had been using the Jazz awareness tooling for at least one year. Studying the use of dashboards and feeds in several large projects using Jazz provides a unique

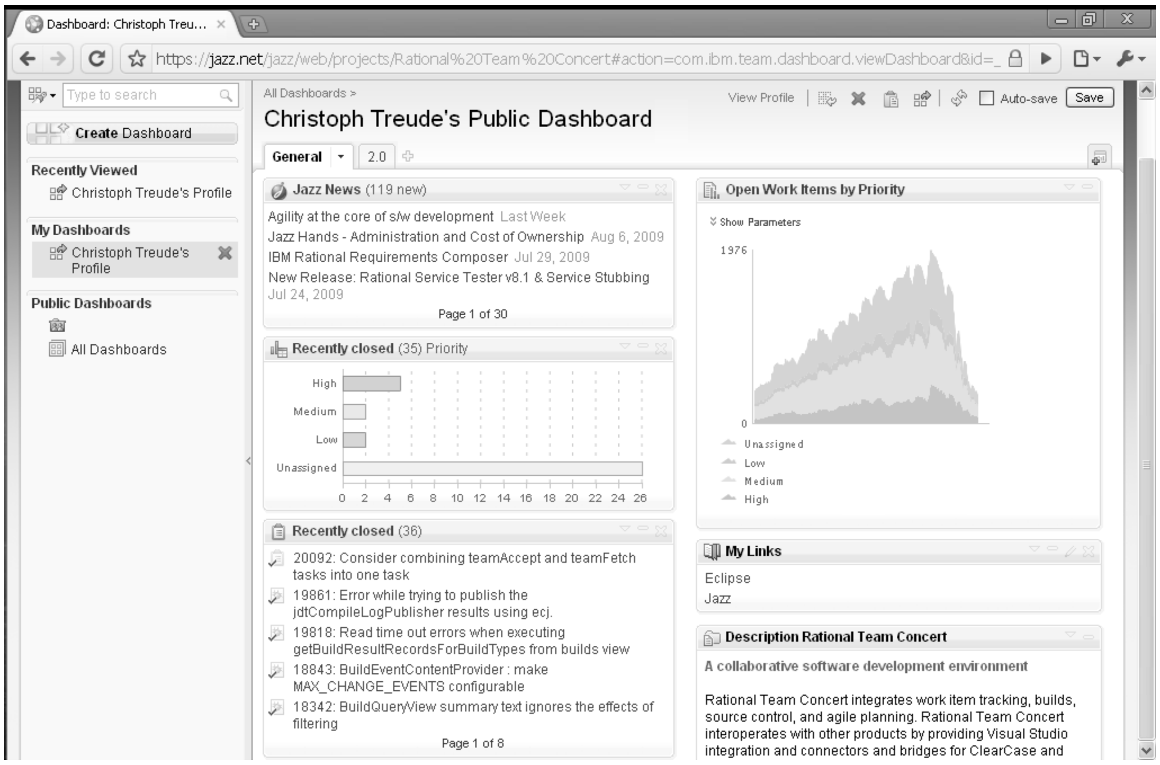
[2]http://cruisecontrol.sourceforge.net/

opportunity to examine the role awareness tools play in software development. As Jazz is one of the first environments to tightly integrate these awareness tools into the IDE, it enables studying the interplay of awareness tools rather than to examine isolated features.

4.1.2 Dashboards and Feeds in IBM's Jazz

In this section, the functionality of dashboards and feeds as implemented in IBM's Jazz is introduced.

Dashboards



Christoph Treude's Public Dashboard

Jazz News (119 new)

Agility at the core of s/w development Last Week
Jazz Hands - Administration and Cost of Ownership Aug 6, 2009
IBM Rational Requirements Composer Jul 29, 2009
New Release: Rational Service Tester v8.1 & Service Stubbing Jul 24, 2009
Page 1 of 30

Recently closed (35) Priority

Recently closed (36)

20092: Consider combining teamAccept and teamFetch tasks into one task
19861: Error while trying to publish the jdtCompileLogPublisher results using ecj.
19818: Read time out errors when executing getBuildResultRecordsForBuildTypes from builds view
18843: BuildEventContentProvider : make MAX_CHANGE_EVENTS configurable
18342: BuildQueryView summary text ignores the effects of filtering
Page 1 of 8

Open Work Items by Priority

My Links

Eclipse
Jazz

Description Rational Team Concert

A collaborative software development environment

Rational Team Concert integrates work item tracking, builds, source control, and agile planning. Rational Team Concert interoperates with other products by providing Visual Studio integration and connectors and bridges for ClearCase and

Figure 4.1: Sample dashboard in IBM's Jazz

Dashboards are information resources that support distributed cognition; they are crucial to many business intelligence applications [54]. Dashboards in the Jazz IDE are displayed and configured using the web interface. They are intended to provide information at a glance and to allow easy navigation to more complete information. By default, each project and each team within a project have their own dashboard;

and an individual dashboard is created for each developer when they first open their web interface.

Figure 4.1 shows an example dashboard. A dashboard consists of several *viewlets*. Viewlets are rectangular widgets displaying information about some aspect of a project. Each viewlet is an instance of a *viewlet type*. The actual content shown in a viewlet depends on the viewlet type (e.g., visual representations of the current workload or a list of members on a team), as well as the way the particular instance has been configured. Developers can add viewlets to their dashboards and configure them using different parameters. While the list of available viewlet types is constantly expanding, Table 4.1 shows common viewlets at the time of this study with a short description. Viewlets can be organized into different tabs within a single dashboard.

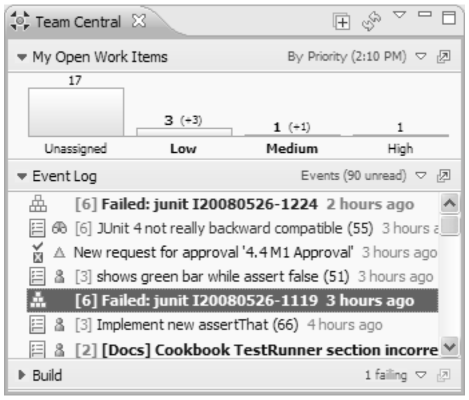
Table 4.1: Viewlet types

type	description
Feeds	internal or external feed
Work Item Statistics	chart of work item query result
Work Items	result of work item query
Reports	pre-defined reports
Bookmarks	customizable list of bookmarks
HTML	snippet of HTML mark-up
Work Item Queries	links to executable queries
About Me	information about a contributor
Builds	notifications from the build engine
Team Members	list of contributors with roles
Plans	progress of plan for an iteration
Description	description of a project or team area
Other	e.g., server status, list of sub-teams

By default, dashboards only display general purpose viewlets containing information about developers and teams, and links to general feeds. In addition to individual customizations of dashboards, project managers or component leads can customize the default settings. It is intended that the development manager of a project is in charge of updating the project dashboard, the component leads are responsible for the team dashboards, and individual developers change their own dashboards.

Feeds

For awareness on the basis of events, Jazz provides feeds. Feeds can either be displayed in the client application or as a viewlet as part of a web-based dashboard. The most common way to view feeds is through the Team Central view in the client application. Figure 4.2 shows an example. Team Central is not accessible through the web interface.



```
[6] Failed: junit I20080526-1224 2 hours ago
[6] JUnit 4 not really backward compatible (55) 3 hours a
New request for approval '4.4 M1 Approval' 3 hours ago
[3] shows green bar while assert false (51) 3 hours ago
[6] Failed: junit I20080526-1119 3 hours ago
[3] Implement new assertThat (66) 4 hours ago
[2] [Docs] Cookbook TestRunner section incorre
```

Figure 4.2: The Team Central view in IBM's Jazz

Team Central is organized into multiple sections that are updated continually with the latest events. By default, Team Central displays a bar chart of current work for the signed-in developer, ordered by priority. The event log in the middle of the view's default configuration shows feeds. These are configured to include build events for all teams that the signed-in developer is part of, work item changes that are pertinent to the signed-in developer, and changes to teams. Developers can add or remove feeds and filter events to personalize their event log. In addition, incoming events are displayed as small pop-up windows in the client application.

Unlike dashboards, feeds and Team Central do not offer functionality for sharing sections, views or events.

4.1.3 Research Methodology

This section identifies the research questions and describes the study setting and data collection methods.

Research Questions

The research questions focus on how and why awareness tools are used, as well as on the impact of awareness tools on software development practices:

1. What is the role of dashboards in supporting individual and collaborative software engineering activities?
   (a) How are dashboards adopted and adapted?
   (b) Why are dashboards used and which roles do they support?
   (c) Which individual and collaborative processes do dashboards support?
   (d) How does the use of dashboards evolve over the lifecycle of a project?
2. What is the role of feeds in supporting individual and collaborative software engineering activities?
   (a) How are feeds adopted and adapted?
   (b) Why are feeds used and which roles do they support?
   (c) How does the use of feeds evolve over the lifecycle of a project?
3. What is the impact of dashboards and feeds on development practices?
4. What are potential tool enhancements?

Research Setting

This study took place with several development teams from IBM. A detailed study was conducted using archival data, semi-structured interviews and a web-based survey with IBM's Jazz team, while the EI teams were only invited to participate in the survey.

Data Collection

The methodology followed a mixed-methods approach, collecting quantitative and qualitative data to allow for triangulation. To gather quantitative data on the configuration of dashboards, the repositories of IBM's Jazz team were accessed and a snapshot of all dashboards along with the viewlets used and their configurations were extracted. 311 dashboards containing a total of 2,975 viewlets were analyzed. At the time of extraction in June 2009, the team was working on the 2.0 release of their product. For the detailed case, qualitative data was collected through a series of interviews and a web-based survey. A total of nine interviews was conducted with the development manager **J-M**, the project administrator **J-A**, one component lead **J-L1** and six developers from five different component teams: **J-D1**, **J-D3**, **J-D4**, **J-D5**, **J-D6**, and **J-D7**.

The web-based survey was given to the developers in the detailed case and the developers in the other teams, and it allowed for considering a significantly larger number of participants compared to interviews. 21 responses were received from Jazz developers (response rate 14%) and 98 responses were received from EI developers (response rate 9%). 76 of the 119 respondents identified themselves as contributors, 28 were component leads, nine were development managers, and six were project administrators.

The questions asked in the survey and interviews were similar. Both the survey and the interview scripts contained about 50 questions on participants' use of feeds and dashboards. Questions inquired about the frequency of use as well as the reasons for using these tools. I investigated which viewlet types were mainly used in dashboards and which sections were mainly used in Team Central, and I inquired about the reasons behind those usage patterns. The questions asked about tool enhancement requests and inquired about the impact of dashboards and feeds. I also sought information about the participants' roles and the teams' development practices. About half of the survey questions required a free form response, while the remaining ones were yes/no or multiple choice questions. Appendix A.2 shows the survey questions. Respondents were allowed to skip questions. In contrast to the survey, the interviews allowed for clarification questions.

In addition, as part of this study, I spent five months on-site frequently having informal discussions with developers regarding their use of dashboards and feeds while facilitating member checking of the findings. The findings gained from the data collection were mirrored in my observations.

4.1.4 Findings

This section presents the study findings organized by the research questions.

RQ1: The Role of Dashboards

In order to answer the first research question, this section outlines the number of dashboards and viewlets to establish the content of a typical dashboard. Then, to further add to the small body of empirical research on awareness, the reasons for dashboard use and the processes they support are discussed.

Adoption and Adaption of Dashboards. 36 of the 74 participants that answered the corresponding question on the survey indicated that they use dashboards, and 38 indicated they do not use them at all. How dashboards are used varies significantly depending on the role of the user. While the project administrator and the development manager in the detailed case study stated that they look at dashboards at least once a day, only 26 respondents were able to recall the last time they looked at a dashboard. Out of these 26 dashboard users, two had used a dashboard within the last 10 minutes before filling out the survey, and another four had used dashboards within the last hour. Seven participants indicated that they had used a dashboard within the last 24 hours, and 13 respondents looked at a dashboard within the last week.

Of the 311 dashboards in the detailed case, two were project level dashboards, 72 belonged to teams, and the remaining 237 belonged to individual contributors. While 121 out of 168 contributors owned only one personal dashboard, there were 47 cases in which contributors created more than one dashboard. That was particularly the case for developers that were part of several teams.

Table 4.2: Number of viewlets per dashboard

# viewlets	# dashboards
>100	1
51-100	3
21-50	19
11-20	38
7-10	66
6	73
5	101
<5	10

The number of viewlets per dashboard is shown in Table 4.2. The majority of dashboards contain five or six viewlets, which is the number of viewlets included in the default configurations (five for contributors and six for teams).

Table 4.3: Most frequently chosen viewlet types, partitioned by dashboard scope

type	project	team	individual	sum
Feeds	7 (24%)	77 (9%)	545 (26%)	629 (21%)
Work Item Statistics	3 (10%)	120 (15%)	298 (14%)	421 (14%)
Work Items	0 (0%)	149 (18%)	168 (8%)	317 (11%)
Reports	6 (21%)	111 (14%)	178 (8%)	295 (10%)
Bookmarks	2 (7%)	18 (2%)	241 (11%)	261 (9%)
HTML	0 (0%)	56 (7%)	188 (9%)	244 (8%)
Work Item Queries	1 (3%)	10 (1%)	176 (8%)	187 (6%)
About Me	0 (0%)	3 (0%)	168 (8%)	171 (6%)
Builds	2 (7%)	73 (9%)	64 (3%)	139 (5%)
Team Members	0 (0%)	70 (9%)	28 (1%)	98 (3%)
Plans	1 (3%)	47 (6%)	32 (2%)	80 (3%)
Description	1 (3%)	65 (8%)	7 (0%)	73 (2%)
Other	6 (21%)	20 (2%)	34 (2%)	60 (2%)
Sum	29 (100%)	819 (100%)	2,127 (100%)	2,975 (100%)

Table 4.3 identifies the most frequently chosen viewlet types, partitioned by the scope of project, team and contributor. Viewlets either show information on project members (e.g., *About Me* and *Team Members*), on artifacts (e.g., *Work Item Statistics* and *Builds*), or on events (e.g., *Feeds*), or they display static content (e.g., *Bookmarks* and *Description*). Overall, the most frequently chosen viewlet types are *Feeds* and *Work Item Statistics*. *Work Item Statistics* display a graphical representation of the results of one work item query, visualized by one parameter. Data can be presented as bar charts, pie charts, tag clouds, or tables. On the project level, *Feeds* and *Reports* are the most common viewlets, while *Work Items* and *Work Item Statistics* account for most viewlets on the team level. Figure 4.1 shows examples of the six most-used viewlet types.

By default, a contributor dashboard displays instructions on how to customize it, an *About Me* viewlet and a *Feed* for a Jazz newsgroup along with two *Bookmarks* viewlets. A default team dashboard shows customization instructions, an *HTML* team description, a *Feed* for team events, the *Team Members*, and *Plans* and *Builds* for the team. These defaults can be adjusted per project and per team, but managers and component leads rarely used that opportunity. Dashboards are individually customized instead. It is impossible to report the default viewlets among the common viewlets as some viewlets may be part of the default configuration, but have been individually customized by developers (e.g., by subscribing to a different feed).

Reasons and Roles for Dashboard Use. The reasons for dashboard use can be categorized into a need for project awareness and for individual convenience.

Gaining a high-level overview of the project status was named 14 times in the survey and in six out of the nine interviews: *"That's what dashboards are for: to give you this sort of overall high-level view of things – it is a faster way than running separate work item queries to see which items are [...] in need of attention."* (J-L1) This high-level overview was found to be particularly useful when looking at releases. In fact, project and team dashboards were often organized by releases, with a separate tab for each release: *"It's really looking at the release at a glance. All the plan items that we have committed to, the open versus closed, some of our burndown, [...] and statistics broken down by team."* (J-A) The ability to produce high-level overviews is especially important for project managers: *"The development manager, that's the perfect person to be using the dashboards, 'cause that's what you want, you want to be reporting things all the time, based on a lot of data."* (J-M)

In addition to the awareness of the project status, dashboards help with awareness of other developers and teams as well as their current focus: *"I then found it also helped to give a peripheral awareness of how the other teams were doing, which I would not have had if I just ran the query for the [my team] items."* (J-L1)

Dashboards also help with the identification of bottlenecks in the development process: *"There were so many things going on late in the game in [this release], that I was relying on that [dashboard] in some respects, too. In the timing meetings, someone would say, 'hey, there's still these items showing up on the dashboard for [a team], is somebody looking at them?' [...] As you're getting towards the end, you're paying more close attention to the last few remaining items as you wind down – and having high-level awareness of what those are and what state they're in is good."* (J-L1) Dashboards are used for *"finding out about work items that need to be fixed or addressed at the end of a development phase."* (J-D6)

For project managers, dashboards provide the opportunity to compare teams against each other and make differences between teams visible: *"One of the reasons I use them a lot is [...] for showing the teams against one another, 'cause it sort of encourages a bit of a competitive thing. Because when you show defects for example, no one wants to be the team that's at the top of the list."* (J-M)

For individual convenience, some viewlets allow for navigation to the underlying queries and work items by clicking on the data. 19 out of the 35 participants in the survey who answered the corresponding question indicated that they use dashboards for navigation: *"I just use that as a shortcut, so I don't have to go and remember the [work item] number."* (J-D4)

As described previously in Section 3.1, sometimes work items are specifically flagged using the tagging feature for work items to increase their visibility and to make sure that they show up on a dashboard: *"There's things that we specifically track. For example, when we were doing the [release] cycle and we wanted to track [a specific issue], we put a* ***`tracking`*** *tag on it, so that it would show up here – it just raised the visibility of it."* (J-M)

Three out of the nine interviewees use dashboards as a work item inbox, in part because of the compact presentation: *"I waste less time navigating through work item queries for things I need to do because they are all on one page."* (J-D5)

When asked if the information displayed in dashboards was otherwise available, all participants agreed that it was, but added that it was much easier to access it through the dashboards: *"Yes, but just not as easily. Because every one of these is based on a query, and all you are really doing is just categorizing things, showing them with a particular view. It's just that you can't get that instantaneous summary of the information, especially if you've got thousands and thousands, that's difficult."* (J-M)

While project and team awareness is more important for project managers and component leads, and individual convenience is the main use case for developers, it is hard to separate these roles – partly because individuals often do not clearly belong to one group or the other. Out of the 76 respondents that identified themselves as contributors in the survey, only 25 stated that they spend 100% of their time on development activities. 27 of the 76 contributors indicated that they spend 50% or less of their time on development activities. At the same time, 13 of these contributors spend 50% or more of their time on project management activities.

Dashboards Support Individual and Collaborative Processes. In Jazz, there are dashboards for three different scopes: project, team and contributor. These dashboards are entirely separate. Dashboards that are used regularly are almost exclusively project and team dashboards. Personal dashboards are usually left in their default configuration, which shows the roles of the developer along with web links and newsgroups. With the current configuration, default dashboards have practically no value beyond indicating team membership.

An interesting case where dashboards support an individual process is the use of feeds viewlets by one developer, considering that feeds are also available in the client application: *"Having that viewlet has saved me a couple of times where I've been wanting to look at the work item inbox from home, and I didn't have the Eclipse client, so my usual path [...] wasn't available to me. So I used the feed viewlet."* (J-L1)

Developers and managers customize dashboards for their own benefit but also do so to communicate important insights to their teams: *"It's not just stuff for me. I often put dashboards up, for things that are not actually things that I care so much about, but they're things that I know the teams will care about – or that I want them to care about."* (J-M)

As dashboards are web pages, it is easy to share them and to send links to dashboards to other project members. This opportunity is used frequently by managers and component leads, using tools such as instant messaging or mailing lists, posting links to dashboard pages on the project wiki, or using dashboards to follow along during planning calls. When communicating the current status of a project to executives, dashboards are also used: *"Often we go to some executive meeting, and they would say – for example, there might be 400 bugs outstanding, and they'd say, 'what insurance can you give us all this stuff is going to be fixed?' and I could say, 'look, in this 3 week cycle we fixed 2,500 bugs, so really, fixing 400 is not a big deal.' "* (J-M)

Dashboards Evolve over the Project Lifecycle. As mentioned before, project and team dashboards often contain release specific information. A prominent example is given by this quote: *"During the [release] cycle, we had this middle column, which is called the mustfix column, which is basically all the defects that have to be fixed for the release. And everything else is, well it's important, but it's not as important as what's in here. [...] And people were constantly looking at those to see who has stuff that needs to be fixed."* (J-M)

This dependence on releases implies that dashboards have to be changed when a product version has been released: *"So now, as we're starting all this new work, this dashboard will probably change a fair bit."* (J-M) Jazz does not offer any means to automate that process – "cleaning up" dashboards is a manual task. However, 25 out of the 36 developers that use dashboards indicated that their use is constant across different project phases. The few exceptions mentioned the "endgame" (i.e., the last cycle of the project before a release).

RQ2: The Role of Feeds

This section looks at a finer granular awareness tool: feeds. Since the data on the use of feeds by developers is not stored on a central server, the findings below are based on the results from the interviews and the survey.

Adoption and Adaption of Feeds. Feeds can be accessed through the Team Central view, through dashboards, and through a couple of other mechanisms in the client application, such as e-mail notifications. 52 out of 86 participants indicated they made use of the Team Central view (see Figure 4.2) and only 10 out of 78 participants that answered this question indicated that they use feeds outside of Team Central. When asked what was the last time they looked at Team Central, nine had looked at it in the last 10 minutes, seven in the last hour, nine in the last 24 hours, 15 in the last week, and 12 did not remember when they had last accessed Team Central.

The feed that most developers look at is the *My Work Item Changes* feed, an event feed of changes to work items that the signed-in developer owns, created, modified, or is subscribed to. Only two developers took advantage of the opportunity to create their own custom feeds (e.g., based on individual work item queries).

Reasons and Roles for Feed Use. Feeds are primarily used to *"track work at a small scale"* (J-D6). The majority of participants who answered the corresponding question in the survey indicated that they use feeds *"to see what work items are updated and new ones coming in quickly and easily."* (J-D5) Then, the work items that are of interest are expanded. Feeds are similar to e-mail notifications in other systems. In fact, a reason to use feeds is because *"it allows [you] to turn off all work item related e-mail."* (J-D5)

The event log in Team Central is largely seen as a personal inbox that is primarily used to answer the question *"What should I do next?"* (J-D6) and thus helps developers plan their day. It also helps to quickly get information such as the due date of a particular feature. The other sections of Team Central, in particular the team load section, are used frequently to get information on the overall status of a team.

In addition, Team Central and feeds are used by new developers on a team to get an overview of what the team is working on and to understand common work practices.

No difference was found in the use of feeds between different development roles. While feeds support collaborative work, how they are used is not shared with other team members.

Feeds Evolve over the Project Lifecycle. The extent to which developers use Team Central is mostly constant across different project phases. Out of 52 responses, only 14 participants in the survey indicated that their use of Team Central was not constant over time. The exceptions were developers who were fairly new to developing software in Jazz and thus were unsure as to how the view worked.

RQ3: The Impact of Dashboards and Feeds

Awareness tools increase the transparency in collaborative software development. As mentioned previously, competition between teams is one of the reasons dashboards are used by managers and component leads. They are aware of the peer pressure effect that arises from this competition, and they see it as one of the benefits of using dashboards: *"That's partly why we use them, they definitely do [create peer pressure]. I mean, it increases the exposure and the visibility a great deal. [...] An executive is probably unlikely to be going in writing a query or browsing around in the work items but it's easy for them just to go here and just right away see."* (J-M) Competition is usually seen as a good thing: *"The need to look like you are making progress is useful."* (J-D6) Any peer pressure is based on data that already exists: *"I mean we're showing data that's already there, it's just making it visible at a glance."* (J-A)

Developers and managers are aware that dashboards only measure quantity not quality. Peer pressure should not occur because of outdated or incorrect data: *"The problem is [that dashboards] are only as good as the data that's in the work items and the way they're being filed, so obviously, if all of this is going to be useful, we need to ensure that the data that's in the open work items is useful, too. [...] If you try to manage purely by numbers, then that's a big mistake. You've got to always be applying common sense."* (J-M) In some cases, correcting steps are taken so that the dashboards and feeds are not misleading: *"Sometimes we have to put those caveats on things. For example, there would be something and would show some team with a really long bar [in a bar chart]. Sometimes there are reasons for that. Just because one team has a lot more defects than another that doesn't necessarily mean that the quality of that component is any worse."* (J-M)

It is worth noting that when developers were asked in the survey if they would alter their work practices because of information displayed in dashboards, 26 out of 32 participants who responded to that question answered no. However, some developers looked at dashboards to compare team activities: *"Just knowing that you're not the component that's on the critical path for number of bugs remaining."* (J-L1)

Whether awareness tools are distracting or not largely depends on their use of push vs. pull mechanisms. None of the respondents found dashboards distracting, in particular because they are not part of the client application but are displayed in a web browser instead: *"You've got to go there to look at them. Of course you could choose not to look at them. I guess one thing that can happen, is just in terms of the amount of noise or information on them – sometimes people go there and they're going looking for one particular thing, and they're seeing a thousand other things going on."* (J-M) Feeds were sometimes considered to be distracting. Six out of 40 respondents found that they receive irrelevant event notifications. On the other hand, some developers consciously *"try to limit unimportant changes and combine them into one change"* (J-D5) to reduce the number of events.

RQ4: Potential Tool Enhancements

This section discusses enhancements for dashboards and feeds that emerged from this study, either from the interviews or from observations which showed that not all developers used the dashboards' full functionality.

Enhancements for Dashboards. The enhancement suggestions for dashboards focus on making them easier to use, and for them to provide more actionable information.

- One of the findings is that developers are often unaware of the full functionality of dashboards. In these cases, this study sometimes helped to raise awareness of features. Two of the interviewees changed their dashboard configuration during the interviews when they learned about the available functionality. While it is difficult to make developers work through the complete list of features, the utility of dashboards could be made more apparent by adding advanced default dashboards. The same applies on a more fine grained level to different settings for particular viewlets. Developers were often unaware of some of the available settings (e.g., visualization and feed configuration settings). Again, default viewlets could help communicate the full utility of the dashboard functionality.

- Developers often miss support for dashboards in the client applications. An integration would enable them to open links to artifacts in the client instead of the web browser.
- More inbox functionality would improve the usefulness of interactions with dashboards. Developers who use dashboards to check events requested that such notifications disappear from the viewlets once read. Also, more information, such as upcoming builds, milestones, feature freezes, or pending approvals, should be easier to access.
- In dashboards with more than one tab, tabs often get missed because they are not obvious to the user. In this case, it would be helpful to show an overview of the available tabs rather than the first page by default or to offer filtering mechanisms, such as only displaying tabs pertinent to a certain release.
- There were also requests to improve the visualizations in dashboards. For example, to compare open vs. closed work items per team, two viewlets need to be created: one for open work items and one for closed work items. Having a stacked chart showing both would save screen real estate and make comparisons easier.
- With the heavy reliance of many project and team dashboards on the release cycle, it should be possible to let viewlets or whole tabs expire when a new version of a product has been shipped. This would help to avoid having outdated information on dashboards.

We have developed a tool – WorkItemExplorer – based on these recommendations, which is described in Section 4.2. In particular, WorkItemExplorer addresses the first suggestion for enhancements by not requiring up-front configuration effort as well as the fourth suggestion on better visualizations by enabling the exploration of relationships between different viewlets. Also, the user interface of WorkItemExplorer is less complex than the dashboard interface, making it easier to understand the tool's full functionality.

Enhancements for Feeds. Most of the enhancement suggestions for feeds and Team Central involve including additional information and perspectives to the tooling.

- A burndown view that shows a graphical representation of remaining work vs. time was requested several times, along with schedules, general announcements, and a section focusing on quality control.
- For work item feeds, it would be useful to identify relationships between tasks that are affected by events. When a sub-task is completed, the tool should show which parent task it belongs to.
- It would be useful to offer more filter options for feeds. In particular, it should be possible to distinguish between items that require action, such as failed builds, and items that do not require any action, such as successful builds. Alternatively, the number of events can be limited by grouping events that belong together. This is already implemented for some cases, but can be more widely applied.
- For some developers, their reason for avoiding feeds in their Team Central view was the time interval for the auto reloads of the event log section. Enabling e-mail notifications would display events faster than the event log would reload.
- Since the layout of Team Central and the configuration of specific feeds are not stored on a central server, they are lost whenever a new version of the client application is installed or whenever a developer switches clients. For many developers, this is a reason not to configure their views.
- Switching between Team Central and dashboards is tedious and could be avoided by having an analog to Team Central in the web interface.

4.1.5 Discussion

This section discusses the findings and identifies future directions for tool designers.

Reliance on Social Media Artifacts for Critical Phases

One interesting finding was the reliance of developers and managers on dashboards in the most critical project phase: the "endgame" right before a release. The project teams in this study used dashboards to stay aware of bottlenecks in the process and work items that needed to be closed before the shipping date. In fact, the defects to be fixed before a release were identified using a combination of social media mechanisms: work items were tagged with `mustfix` to indicate their importance, and then dashboards and feeds were used to track the state of those work items. In one of the dashboards, a `mustfix` column was implemented that only showed viewlets based on the `mustfix` tag, and feeds were used to see state changes of the `mustfix` items in real-time. This reliance on social media tools demonstrates the versatility and the reliability of social media artifacts in software development. Social media tools can be used to organize artifacts in an informal manner and to support tasks that do not occur every day.

Relationship between Dashboards and Feeds

There are three main differences between dashboards and feeds: granularity, privacy and configurability. Even though it is possible to make dashboards private, most developers share dashboards with the whole team. Project and team dashboards are shared by default with the entire team. This is in contrast to feeds in the client application: *"In Team Central it's a bit too introverted, because it's your own view of things, and you can't share that with people."* (J-A)

The suggestions for tool enhancements indicate that the relationship between dashboards and feeds is not clear-cut. It was suggested that dashboard functionality be introduced into Team Central, in particular burndown views and grouping of events. At the same time, developers requested more inbox functionality for dashboards, such as viewlets for incoming work items and work item changes.

Implementing these suggestions would result in Team Central being very similar to dashboards and vice versa. There are two different conclusions possible: a clear distinction between dashboards and Team Central, or a tight integration. As J-D1 put it, *"there seems to be a split between managers and non-managers"* with regard to their tool use. This border could be sustained, and developers could be required to use dashboards as soon as they are looking for information outside their own work items. Alternatively, dashboards and Team Central could consciously be aligned to have similar functionality and configurations. This could go as far as changes to sections in Team Central being reflected in dashboards and vice versa. In the current setting, individual dashboards do not have obvious uses, unlike team and project dashboards. Individual dashboards often remain unchanged, display outdated information, such as links to feeds with expired passwords, and are rarely used.

The current distinction between overall project status in dashboards and pertinent work item changes in the client application is insufficient as developers and managers need both: awareness in the large and in the small, but to different extents. An example are the `mustfix` work items: developers not looking at the project dashboard regularly had to rely on being informed by somebody of the `mustfix` tag in order to adjust their priorities. The results indicate that integrating web-based dashboards and Team Central functionality would solve some enhancement requests and provide a clearer conceptual model of awareness.

Dashboards and feeds are ephemeral resources in a software development environment. The secret of their success may lie in the fact that often team memberships and roles turned out to be ephemeral as well. When developers need to take on tasks of project managers and vice versa, a strict distinction between high-level awareness and low-level awareness is rendered ineffective.

The Paradox of Choice

As with many highly configurable user interfaces, dashboard users suffer from the paradox of choice. Most developers are unaware of all the options available to them. This leads to unsatisfactory configurations and to developers avoiding dashboards due to not understanding their benefits. Instead of adding more viewlet types and properties, removing the least used viewlet types and offering well-designed default configurations would be a better solution.

Towards Development Environments as Socially Translucent Systems

In their work on socially translucent systems, Erickson and Kellogg suggest that systems that support social processes have three characteristics: visibility, awareness and accountability [58]. Development environments are systems that support social processes in software development; and thus need to be designed with these characteristics in mind. These issues were clearly evident in the findings on how dashboards and feeds provide different levels of awareness in software development. Many enhancement requests contradict the original intent of the individual features. When combined, the enhancement requests point to the need for one integrated system rather than a set of isolated features to improve awareness and visibility.

4.1.6 Limitations

For usage data on dashboards and feeds, I had to rely on answers from the interviews and the survey. Jazz does not log data on the use of these artifacts. However, a strength of the mixed-methods approach is the possible triangulation of findings obtained through different methods. The large number of respondents in the survey also provides a good basis for the conclusions.

Some of the individuals who participated in the survey skipped some of the questions while filling out the survey. Consequently I reported throughout this section the number of responses to the questions discussed. Another limitation lies in the low response rates to the survey. However, as it is the nature of a large software project that developers do not have much time to spare and since for ethical reasons, I did not want the developers to feel coerced to participate, it was impossible to achieve better response rates.

When the teams started on their projects, dashboards were still being improved and new viewlet types were added. This might have influenced the specific usage. Because the detailed case study was conducted with the developers of Jazz, their willingness to adopt dashboards and feeds might be above-average. However, the findings were confirmed with developers from EI. While the team in the detailed case used an agile development method, not all of the additional teams were agile. No differences were found between users across the different teams and development methods.

4.1.7 Summary

Awareness in a software development project is composed of many different aspects: developers need to stay aware of the overall status of their tasks and critical deadlines, they need to understand current priorities and bottlenecks in the process, they need to know of dependencies between components and teams, and they need to be informed of changes to tasks they are working on in a real-time manner. Previous efforts have often neglected high-level awareness and focused on source code or source code management systems instead. While the awareness of source code is crucial to software development, it is insufficient when development is a team effort. Developers need to be able to see current plans, upcoming deadlines, schedules of other team members, and critical components.

This research has shown how several teams of developers using IBM's Jazz use a combination of feeds and dashboards to maintain awareness of various aspects of their task management system: dashboards are mainly used to keep track of the overall project status, to provide peripheral awareness of the work of other teams and to stir competition between teams. During critical project phases, dashboards evolve into the central entity where priorities of tasks are organized and shared. Therefore, dashboards depend largely on the project's lifecycle. Feeds are used to track work at a small scale and to plan short-term development activities.

As developers need to understand issues that span several teams and components, the line between requirements for dashboards and feeds becomes blurred and tool enhancement requests lead to a considerable overlap of both tools. These findings suggest that such a toolset brings awareness of projects, teams and tasks together and no longer makes distinctions between project-wide awareness (i.e., awareness in the large) and awareness related to the immediate tasks a developer is working on (i.e., awareness in the small).

Also, there is a need for informal tool support for informal processes. Tool support inspired by social media, such as tags and feeds, are promising approaches towards fulfilling this need. The next section presents WorkItemExplorer, a tool we developed based on the findings described here.

4.2 WorkItemExplorer

To address some of the tool enhancement suggestions identified in the previous section, we have developed WorkItemExplorer, an interactive environment to visually explore data from software development tasks. WorkItemExplorer enables developers and managers to investigate trends and correlations in their task management system by making data exploration flexible and interactive.

Unlike dashboards, WorkItemExplorer does not require up-front configuration effort to show useful information, its user interface is simpler, and it enables the exploration of relationships between different visualizations by leveraging multiple coordinated views. Multiple coordinated views allow users to have multiple different views, such as bar charts or timelines, open at the same time, all displaying the same data in different ways. The coordination comes into play when interacting with the views; highlighting one data element will have a mirrored effect on all other views. This enables the exploration of data relationships as well as the discovery of trends that might otherwise be difficult to see because they span multiple aspects. Work-

ItemExplorer is a web-based tool built on top of the Choosel framework[3]. We have implemented an adapter for queries against the work item component of IBM's Jazz platform, but other task management systems could be integrated in the future.

We evaluated WorkItemExplorer in three ways: we conducted a usability study with four post-doctoral participants to verify the usability of the tool functionality, we conducted an evaluation with four members of IBM's Jazz team, and we manually verified that WorkItemExplorer is able to answer the questions developers ask as identified by Fritz and Murphy [65]. Our evaluation yields three main findings:

- WorkItemExplorer can answer questions that developers ask about task management systems,
- WorkItemExplorer enables the acquisition of new insights through the free exploration of data, and
- WorkItemExplorer offers a flexible and tactile environment in which different individuals solve the same task in different ways.

The remainder of this section is structured as follows. The tool is introduced in more detail in Section 4.2.1, and example scenarios for the tool are described in Section 4.2.2. Section 4.2.3 presents the results from our evaluation before summarizing this work in Section 4.2.4.

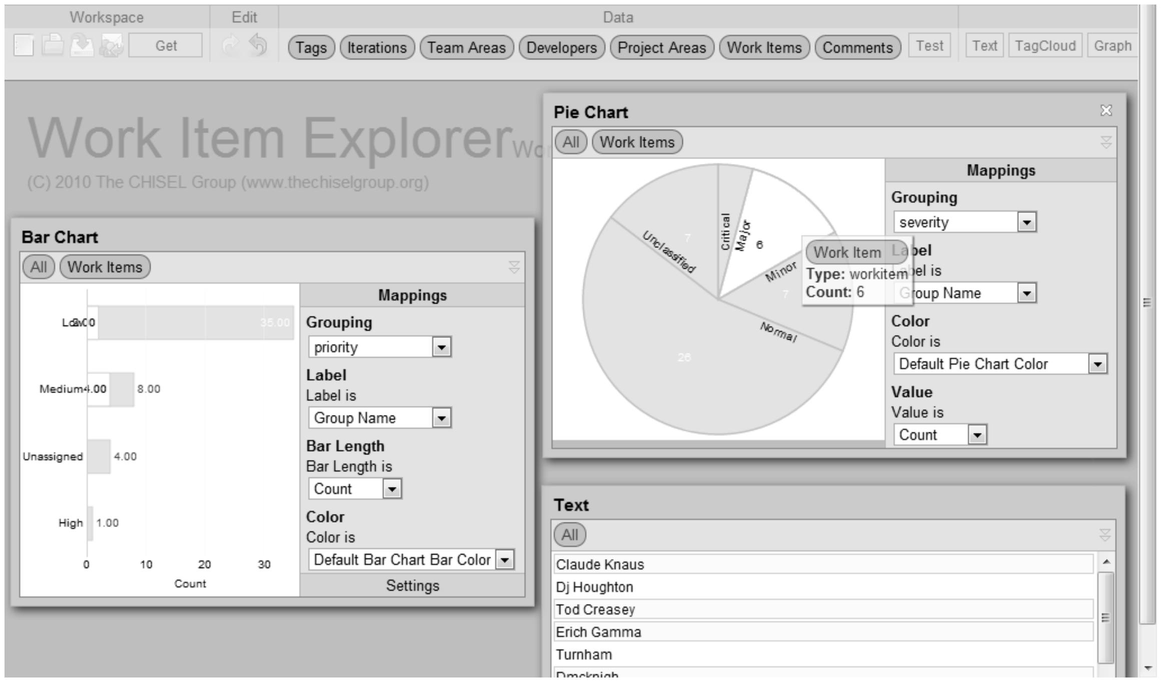
4.2.1 The WorkItemExplorer Tool

WorkItemExplorer is an interactive tool that allows users to dynamically explore data from a software development task management system. As shown in Figure 4.3, users can have multiple views open at the same time, all displaying the same data in different ways. In addition, highlighting a data element in one view will have a mirrored effect in all other views.

WorkItemExplorer supports seven data elements and the relationships between them: work items, developers, iterations, project areas, team areas, tags, and comments. Using a drag-and-drop interface, these data elements can be moved into seven different views:

- A **text** view with different grouping options (e.g., to see a list of work items grouped by their owner).

[3]http://thechiselgroup.org/choosel



Bar Chart — Mappings
Grouping: priority
Label is: Group Name
Bar Length is: Count
Color is: Default Bar Chart Bar Color
Settings

Pie Chart — Mappings
Grouping: severity
Label is: Group Name
Color is: Default Pie Chart Color
Value is: Count

Work Item
Type: workitem
Count: 6

Text
Claude Knaus
Dj Houghton
Tod Creasey
Erich Gamma
Turnham

Figure 4.3: Screenshot of WorkItemExplorer

- A **tag cloud**, primarily for the exploration of work item tags.
- A **graph** for the exploration of relationships between different kinds of artifacts, such as work items and iterations. Expanding a relationship on any note in the graph will add the corresponding notes to the view.
- A **bar chart** to visualize data with different groupings using bars of different lengths (e.g., to visualize developers by the number of work items they own).
- A **pie chart** to visualize data with different groupings using pie wedges of different sizes (e.g., to show work items by priority).
- A **heat bars** view to visualize work items over time, with an additional grouping option (e.g., to visualize the creation of different work item types, such as defects and enhancements, over time). Heat bars are based on ConcernLines (see Section 3.2).
- A **timeline** to analyze data over time. Different time properties, such as creation or modification date, can be chosen (e.g., to visualize team area creation over time).

4.2.2 Example Scenarios

In this section, two scenarios are described that highlight the functionality of WorkItemExplorer.

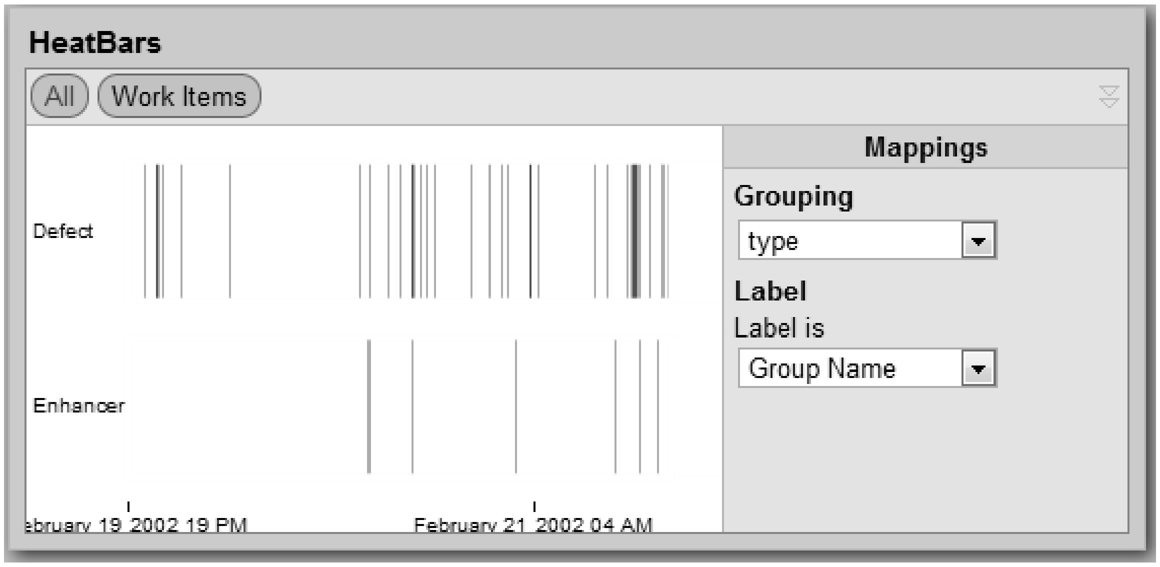


Figure 4.4: Screenshot of the heat bars view of WorkItemExplorer

Who is Working on Important Work Items?

To identify process bottlenecks or to balance workload, it is crucial to know who works on important tasks. The task importance can be defined in many different ways, and WorkItemExplorer can be used to understand the implications of different approaches. For the purpose of this scenario, we assume that task importance is defined by high priority and severity. To explore important work items and their owners, users can open up a bar chart and a pie chart in WorkItemExplorer, and then drag all of the work items onto both of them. They can then group the bar chart by priority, and the pie chart by severity, and then drag the bars and pie wedges that they are interested in into a third view, such as a text view. For example, users could drag the bars and pie wedges corresponding to *high* priority and *major* as well as *critical* severity. If they now group the text view by work item owner, users get a list of all people working on important work items (see Figure 4.3), and they can continue to explore their workload in more detail.

In addition, this configuration allows us to immediately explore the relationship between severity and priority of work items in our data set. As shown in Figure 4.3, the user is exploring the correlation between severity and priority of work items using a bar chart that shows work items grouped by priority, and a pie chart grouped by severity. Since the user is hovering over the pie wedge for major severity, all corresponding items are highlighted in all views. The text view shows the owners of the corresponding work items.

What Work Items are Created When?

Understanding when different kinds of work items, such as defects and enhancements, are being created can be very useful to ensure that a team is correctly following a particular process. To find this kind of information, WorkItemExplorer features a heat bars view that visualizes work item creation grouped by one property, such as type or priority, over time. Figure 4.4 shows the result for a grouping by work item type, and we can see that enhancements were added near the end of the data set, whereas defects were added constantly throughout with a few more intense periods. To further explore this phenomenon, users could now drag either of the bars into another view for further analysis.

4.2.3 Evaluation

We conducted a usability study as well as an evaluation with IBM's Jazz team, and we found support that the tool is able to answer a broad range of questions.

Usability Study

We recruited four post-doctoral researchers from our research group **P-1**, **P-2**, **P-3**, and **P-4** to evaluate the usability of WorkItemExplorer. A data set containing the first 48 work items along with all related artifacts from the Eclipse Bugzilla bug tracker[4] was used for the study. After a brief introduction to the tool, the participants were given a list of five tasks that are based on the questions that developers ask identified by Fritz and Murphy [65]:

1. Try to find out which work items your team members are working on.[5]

[4]https://bugs.eclipse.org/bugs/

[5]We gave two specific names from our data set as "team members".

2. Who is the busiest developer at the moment? If we were to reassign some of their work items, who should we give them to?

3. You need to find a new work item to work on. Try to find work items that are of high priority and severity to complete next.

4. You are currently interested in work items with the tags *investigate* and *need-info*. Try to find the comments on those work items.

5. Please explore the data on your own and let us know of anything interesting that you find.

The upper half of Tables 4.4 to 4.8 shows the results of the usability evaluation[6]. The first four tasks were completed in less than five minutes per task with some participants solving tasks in less than 30 seconds, and only three participants were unable to solve one of the tasks (denoted in bold). Participants used different views to solve the same tasks. This is an indicator that there are many different ways to gain insights using WorkItemExplorer, allowing for a broad range of insights as well as serendipity. For example, when switching from Task 1 to Task 2, P-3 realized that they had already produced the answer on the screen as part of Task 1.

Feedback on the tool was generally positive: *"very usable"* (P-1), *"the best part is that I can click, select stuff and move it and see what it looks like in another view"* (P-3), and *"very cool interface"* (P-4). Also, all participants used the opportunity for free data exploration as part of Task 5.

Evaluation with IBM's Jazz Team

After the usability study, we recruited four members of IBM's Jazz team as participants: the development manager **J-M** as well as a component lead **J-L1** and two developers from two separate component teams **J-D1** and **J-D8**. The lower half of Tables 4.4 to 4.8 shows the results of the evaluation. Like the post-doctoral researchers, the developers used different views to solve the same tasks, and apart from the development manager J-M, the participants were able to solve their tasks in less than two minutes.

Additionally, in our evaluation with IBM's Jazz team, we were able to have the participants explore data from their own task management system. To do so, the

[6]We slightly adjusted the wording of Tasks 3 and 4 after P-1.

Table 4.4: Completion time and views for Task 1: "Try to find out which work items your team members are working on."

participant	time	views
P-1	1:35	Graph, Text
P-2	1:35	Text, Graph
P-3	1:05	Bar, Text
P-4	4:50	Text, 2 Graph
J-M	**3:23**	**Bar, Graph, Text**
J-L1	0:51	Bar, Text
J-D1	1:17	Text, Graph
J-D8	4:58	3 Text, Bar, Graph

Table 4.5: Completion time and views for Task 2: "Who is the busiest developer at the moment? If we were to reassign some of their work items, who should we give them to?"

participant	time	views
P-1	2:35	2 Bar, Text
P-2	1:50	Graph, Tag, Text
P-3	0:20	
P-4	3:15	2 Bar, Pie
J-M	**2:07**	**Bar**
J-L1	0:26	Bar
J-D1	1:01	Graph, Bar
J-D8	1:09	2 Bar

participants had to run a work item query against their task management system to import data into WorkItemExplorer. Then, they could use the tool to explore their data. Table 4.9 shows the queries that participants ran. J-M explored all work items that had been tagged with the `tracking` tag (i.e., work items that he had previously tagged because they were important to him). J-L1 queried for a particular feature that he had been involved in, and J-D8 queried for work items related to his team. In addition, J-D1 explored work items that had been created in the last two days. These queries point at interesting use cases for WorkItemExplorer – to explore work items related to features, teams, or tags.

The feedback we received was generally positive, particularly when comparing WorkItemExplorer to the functionality that developers use right now, such as dashboards and work item queries: *"this a lot easier to do [than dashboards]"* (J-D8) and *"when you are constructing a work item query it feels more like a data entry task, whereas this feels more like you are browsing and exploring the space, and it's almost*

Table 4.6: Completion time and views for Task 3: "You need to find a new work item to work on. Try to find work items that are of high priority and severity to complete next."

participant	time	views
P-1	3:45	Bar, Text
P-2	2:10	Bar, 2 Text
P-3	1:05	Bar
P-4	2:20	2 Bar, Graph, Text
J-M	**4:00**	**3 Bar, Graph, Text**
J-L1	3:03	4 Bar, 2 TagCloud
J-D1	0:50	Bar
J-D8	1:43	2 Bar

Table 4.7: Completion time and views for Task 4: "You are currently interested in work items with the tags *investigate* and *needinfo*. Try to find the comments on those work items."

participant	time	views
P-1	**6:50**	**7 Text**
P-2	1:05	Tag, Graph
P-3	2:40	Bar, 2 Tag, Text, Graph
P-4	**4:55**	**3 Text, Bar, 3 Graph**
J-M	**6:02**	**2 Graph, 2 TagCloud, Text**
J-L1	3:55	TagCould, Text, 3 Bar, Graph
J-D1	1:25	Text, TagCloud, Graph
J-D8	1:37	2 Text, Graph

more tactile" (J-L1). In addition, J-L1 compared the tool to faceted browsing: *"[This has a] similar flavour to faceted browsing, [...] the idea of this whole space that you are exploring and being able to slice and dice it in various ways. [Then you can] navigate down to the bits that you are interested in."*

Questions Developers Ask

To ensure that WorkItemExplorer can answer questions developers ask in their daily work, we manually inspected the 78 questions that developers ask, but for which support is lacking, as identified by Fritz and Murphy [65]. 17 of these questions are applicable to WorkItemExplorer; the remaining 59 require data beyond what is currently included in WorkItemExplorer, such as builds or source code. However, we did confirm that WorkItemExplorer is able to answer all 17 questions that are focused on task data. Appendix B.1 contains all 78 questions along with an indication as to

Table 4.8: Views for Task 5: "Please explore the data on your own and let us know of anything interesting that you find."

participant	views
P-1	HeatBars, Text, Timeline
P-2	2 Text, Graph
P-3	5 Bar, Text, 2 Tag, Graph
P-4	Graph
J-M	Text, Bar, Pie, Graph
J-L1	2 Bar, 2 Text
J-D1	TagCloud, 3 Text, 2 Graph, Pie
J-D8	HeatBars, Text, Bar

Table 4.9: Queries used by study participants

participant	project	summary	creation date	modification date
J-M	ALM	"Track"	N/A	current year
J-L1	RTC	*[feature]*	*N/A*	*N/A*
J-D1	RTC	*N/A*	last two days	*N/A*
J-D8	RTC	*[team]*	last two weeks	*N/A*

whether WorkItemExplorer can answer them.

Limitations

The evaluation we conducted is preliminary and the generality of our findings is limited, in particular because we only had four professional participants. However, we collected both qualitative and quantitative data in our evaluation to strengthen our methodology, and we found preliminary evidence that WorkItemExplorer can help developers in managing their tasks.

For any specific question, there might be other tools that can provide an answer more quickly than WorkItemExplorer. However, the strength of an exploration tool lies in the free exploration of data when no goal is specified, and in the power to provide serendipitous insights.

4.2.4 Summary

With WorkItemExplorer, we have introduced an interactive environment that enables developers and managers to visually explore data from software development task management systems by leveraging multiple coordinated views. Our evaluation has shown that WorkItemExplorer is able to answer questions that developers ask, and we have confirmed the usability of the tool through a study with professional developers. The development of WorkItemExplorer is a step towards making awareness tools more accessible to those stakeholders in a software development project who are not full-time managers. The essential quality of the tool is its flexibility and its informality. Unlike dashboards, views in WorkItemExplorer do not need to be configured up-front, and rather than making views and data persistent, the tool's configurations are ephemeral. In addition, the tool emphasizes the role of social media artifacts, such as tags, by making them central to the data exploration.

This section concludes the findings with regard to task management by teams using IBM's Jazz. Social media artifacts, such as tags, dashboards, and feeds, play an important role in collaborative software development processes, they have been adopted and adapted by professional software developers to suit their needs, and they have become an essential part of many processes, such as lifecycle management and awareness. With ConcernLines and WorkItemExplorer, two tools have been introduced that aim at leveraging the knowledge available in social media artifacts by promoting them to central artifacts in a development tool.

The next section examines the role of social media artifacts in task management by teams using Google Code as their development platform.

Chapter 5

Task Management using Google Code

After studying the role of social media artifacts in the task management of development teams using IBM's Jazz, this chapter examines the role of social media artifacts – particularly labels – in the task management of development teams using the Google Code Issue Tracker, a free hosting service for open source projects provided by Google[1]. For each project on Google Code, project members can configure their own setup, and Google provides them with a source code repository, a task management system[2], wiki pages, and download pages.

Studying task management on Google Code is beneficial for several reasons. First, the particular features of Google Code offer teams flexible ways to configure their task management systems. Through empirical studies, we can determine what works well and what does not work well, and we can make recommendations on approaches to task management that are used by development teams in practice. Second, as an open source project hosting site, Google Code has a large number of projects, and we can base our findings on many different development teams, and compare their approaches to task management.

At the time of the study, several thousand projects were using the project hosting service, with Google's own Android project[3], the Google Web Toolkit (GWT)[4] and

[1]http://code.google.com/p/support/wiki/GettingStarted

[2]On Google Code, the task management system is called "issue tracker". For the purposes of this thesis, there is no difference between task management systems and issue trackers.

[3]http://code.google.com/p/android/

[4]http://code.google.com/p/google-web-toolkit/

Chromium (the open source project behind Google Chrome)[5] being among the most prominent.

5.1 Labelling on Google Code

Google Code allows its users to customize their task management system by defining custom task categories, setting default values, surfacing different task properties, tagging tasks, and creating categories on the fly. This is done through labels, with Google Code distinguishing between two different kinds of labels. Basic *labels*, such as `Performance`, are identical to the work item tags in IBM's Jazz. However, Google Code goes beyond these basic labels to support *key-value pairs*. Key-value pairs are labels that contain one or more dashes in their name, e.g., `Priority-High`. The part before the first dash is the key (`Priority` in the example), and the part after that dash is considered to be the value (`High` in the example). While this functionality is similar to the pre-defined categories in IBM's Jazz task management system with categories such as priority and severity, it offers more flexibility because the keys are not pre-defined, and new keys can be added in the same way that new labels are added – simply by attaching a label with a dash to a task[6].

The remainder of this section is structured as follows. In Section 5.1.1, the labelling feature of Google Code is introduced in detail. Then, the research methodology for this study is described in Section 5.1.2, and the findings are presented in Section 5.1.3. The findings are discussed in Section 5.1.4, and the limitations of the study are described in Section 5.1.5, before summarizing the results in Section 5.1.6.

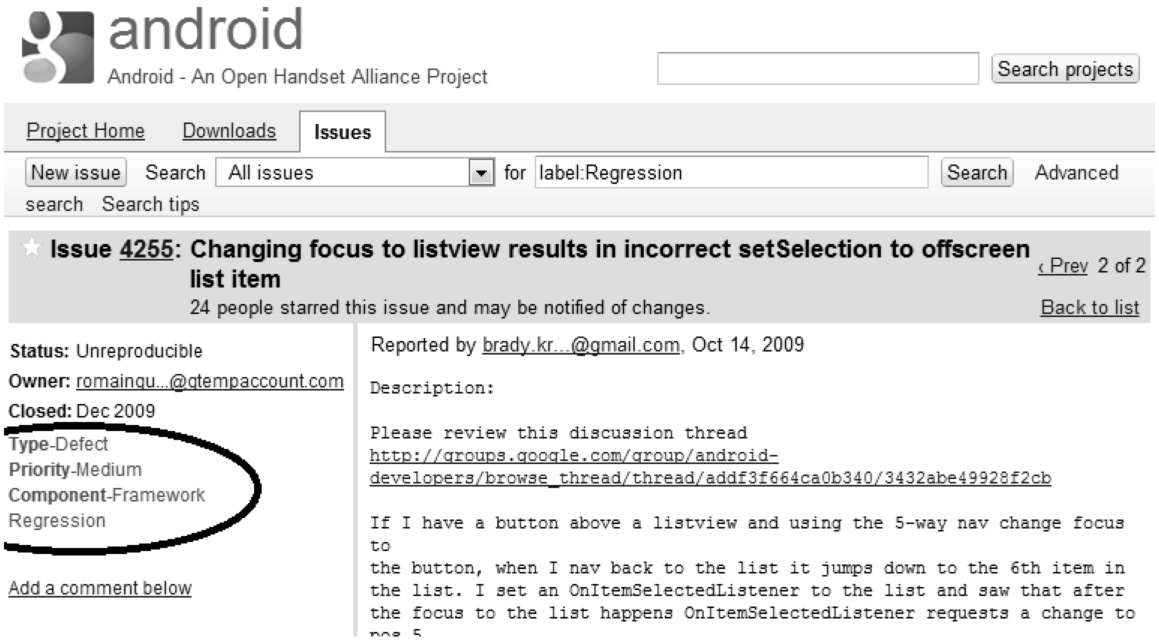
5.1.1 Task Labels on Google Code

Figure 5.1 shows an example of a task from Google's Android project. The task has been labelled with `Regression` to indicate that the problem described in the task was found during regression testing[7]. In addition, the task was annotated with three key-value pairs to indicate its type, priority, and component: `Type-Defect`, `Priority-Medium`, and `Component-Framework`.

[5]http://code.google.com/p/chromium/

[6]http://code.google.com/p/support/wiki/IssueTracker

[7]Regression testing is defined as the re-running of test cases to ensure that software changes have no unintended side-effects.



Issue 4255: Changing focus to listview results in incorrect setSelection to offscreen list item
24 people starred this issue and may be notified of changes.

Status: Unreproducible
Owner: romaingu...@gtempaccount.com
Closed: Dec 2009
Type-Defect
Priority-Medium
Component-Framework
Regression

Reported by brady.kr...@gmail.com, Oct 14, 2009

```
Description:

Please review this discussion thread
http://groups.google.com/group/android-
developers/browse_thread/thread/addf3f664ca0b340/3432abe49928f2cb

If I have a button above a listview and using the 5-way nav change focus
to
the button, when I nav back to the list it jumps down to the 6th item in
the list. I set an OnItemSelectedListener to the list and saw that after
the focus to the list happens OnItemSelectedListener requests a change to
```

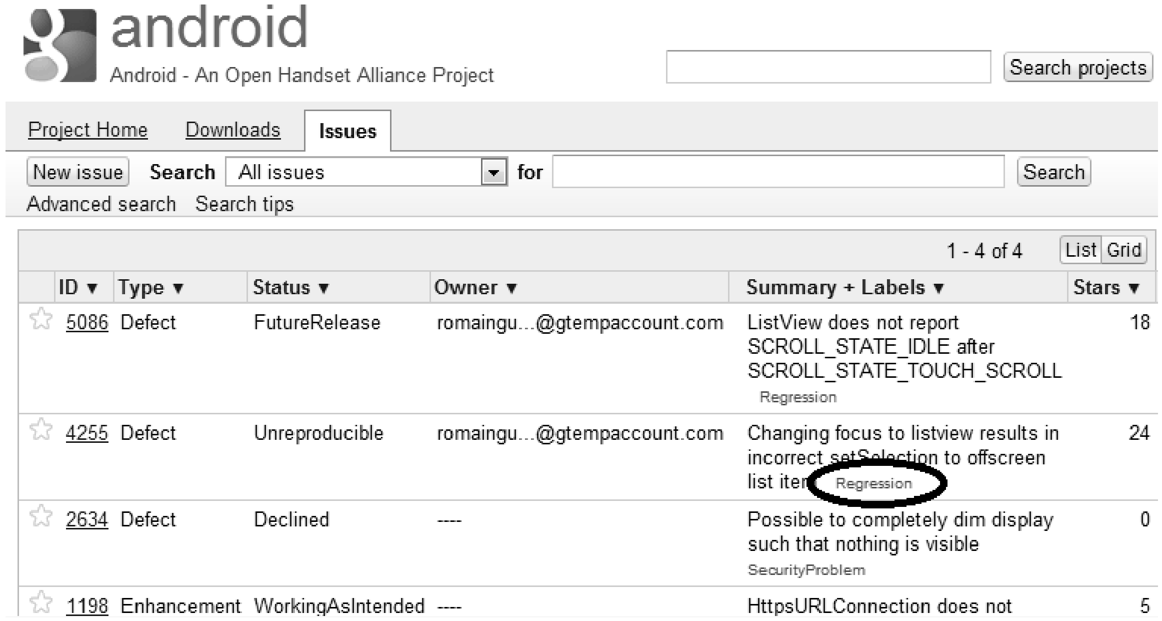
Figure 5.1: Task on Google Code with key-value pairs and labels

On Google Code, the project owner can specify a list of pre-defined labels and key-value pairs along with their exact meaning (that will be displayed on mouse-over in the task management system). The owner can also specify whether certain key-value pairs should be treated as single-valued or multi-valued. For example, the owner might designate that there can only be one key-value pair starting with `Priority-` per task to avoid conflicting task priorities, whereas it might be feasible to have one task belong to several components as indicated by several `Component-` keys. Two key-value pairs (`Priority-Medium` and `Type-Defect`) are suggested by Google Code as default for all new tasks.

Key-value pairs also play a role in the display of all tasks in a tabular form that is shown every time a developer accesses the task management system (see Figure 5.2). All tasks are displayed as a table where each row corresponds to a task, and each column corresponds to a key. The list of keys to be displayed is freely configurable. As shown in Figure 5.2, basic labels are displayed as part of the summary column.

5.1.2 Research Methodology

This section describes the research methodology used by presenting the research questions as well as the methods for data collection and analysis.


android
Android - An Open Handset Alliance Project

Search projects

Project Home Downloads Issues

New issue Search All issues for Search
Advanced search Search tips

1 - 4 of 4 List Grid

ID	Type	Status	Owner	Summary + Labels	Stars
5086	Defect	FutureRelease	romaingu...@gtempaccount.com	ListView does not report SCROLL_STATE_IDLE after SCROLL_STATE_TOUCH_SCROLL Regression	18
4255	Defect	Unreproducible	romaingu...@gtempaccount.com	Changing focus to listview results in incorrect setSelection to offscreen list item Regression	24
2634	Defect	Declined	----	Possible to completely dim display such that nothing is visible SecurityProblem	0
1198	Enhancement	WorkingAsIntended	----	HttpsURLConnection does not	5

Figure 5.2: Task table on Google Code with labels

Research Questions

The research questions focus on the mechanisms that developers have available to them for task management on Google Code.

1. Which mechanisms do projects use to manage their tasks?

2. What labels and key-value pairs do projects use for task categorization?

3. What characteristics of labels and key-value pairs are prevalent?

4. Which keys are surfaced in the task management system?

Data Collection

The complete task management data for a sample of 1,000 different projects from Google Code was downloaded. Google's own search algorithm was used to determine the 1,000 most active projects by sampling the first 1,000 projects (out of more than 5,000) that appear on the Google Code project search[8].

For each project, all tasks were retrieved, as well as all the labels and key-value pairs that were attached to them. For each task, I extracted its ID, title, description,

[8]http://code.google.com/hosting/search?

owner, state, as well as when it was last updated, if and when it was closed, and how many times it was marked with a “star” by a developer. The columns that each project displays by default were also extracted to be able to understand which of the keys are surfaced in the task management system. For the labels and key-value pairs, the project and task they belonged to were recorded as well as the text of the label or key-value pair, respectively. All data was stored in a relational database.

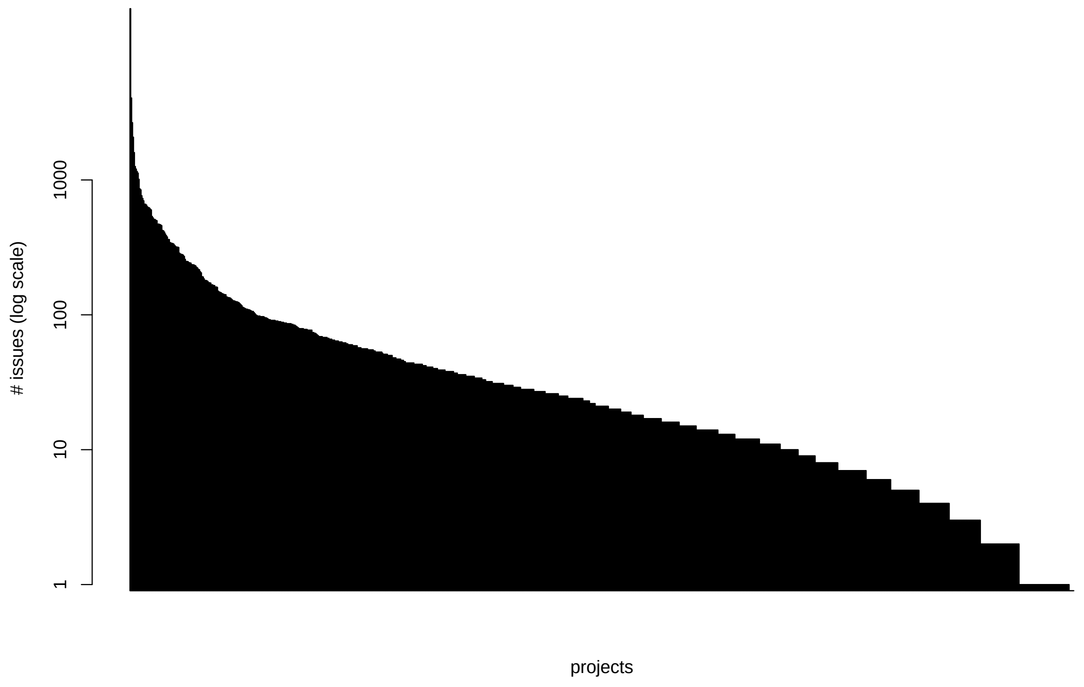


Figure 5.3: Distribution of tasks to projects

In total, 91,270 tasks belonging to 1,000 different projects were extracted. The projects with the most tasks were Android (18,630 tasks), CyanogenMod (4,058 tasks), and Yii (2,662 tasks). Figure 5.3 shows the distribution of tasks to projects using a log scale. In addition, 7,562 instances of a label being attached to a task were extracted, as well as 226,578 instances of a key-value pair being attached to a task.

Data Analysis

To answer the quantitative research questions regarding what mechanisms different projects use to manage their tasks, I ran queries against the relational database. To answer the qualitative research question on the characteristics of labels and key-value pairs, the same coding scheme that was used for tags in IBM's Jazz was adopted (see Section 3.1). As with the previously described study, the "heads" of the "long tails" from the distribution of label instances to label keywords and from the distribution of key-value pair instances to keys were sampled. In the case of labels, the 83 most-used labels in the data set were coded (all labels with at least 16 instances). These 83 label keywords accounted for 80% of all label instances in the data. In the case of key-value pairs, the 34 most-used keys were coded (all keys with at least 95 instances). These 34 keys accounted for 99% of all instances of key-value pairs[9].

Based on the coding scheme developed in the previous chapter, each of the labels and each of the keys in the sample were coded. During the coding process, all data available on Google Code was taken into account. While some of the labels and keys were self-explanatory, for others, the project wiki or the task descriptions were considered.

5.1.3 Findings

This section presents the findings on the role of labels in the task management of software development teams that use the Google Code Issue Tracker.

RQ1: Mechanisms for Task Management

Table 5.1: Mechanisms for task management used by different projects

project	labels		key-value pairs		sum
Android	26	(0.04%)	58,417	(99.96%)	58,443
CyanogenMod	1,318	(11.50%)	10,144	(88.50%)	11,462
Yii	6	(0.10%)	5,728	(99.90%)	5,734
cocos2d for iPhone	26	(0.64%)	4,041	(99.36%)	4,067
DataObjects.Net	352	(9.19%)	3,480	(90.81%)	3,832
TarreWoW	0	(0.00%)	3,777	(100.00%)	3,777
Modellus	20	(0.57%)	3,466	(99.43%)	3,486
Transfer.nu	0	(0.00%)	3,238	(100.00%)	3,238
Quick Search Box	16	(0.55%)	2,898	(99.45%)	2,914
Gallio Automation Platform	78	(2.92%)	2,594	(97.08%)	2,672

To answer the first research question on what mechanisms different projects use to manage their tasks, the number of labels and the number of key-value pairs were

[9]I chose not to focus on the keys corresponding to 80% of the key-value pair instances as there were only three keys (`Priority`, `Type`, and `Milestone`) responsible for these instances.

determined for each project in the data. Table 5.1 shows the results for the 10 most active projects. All of these projects made heavy use of the key-value pairs, and two of the projects (TarreWoW and Transfer.nu) used key-value pairs exclusively. On the other end of the spectrum, in some projects, such as CyanogenMod and DataObjects.Net, labels accounted for roughly 10% of the task management mechanisms applied.

RQ2: Most-Used Labels and Key-Value Pairs

For the second research question on the most-used labels and key-value pairs, the most-used items across all 1,000 projects were calcuated, and a more detailed analysis of the two most active projects in the data was conducted: Android and CyanogenMod. Android is an operating system for mobile devices, such as smartphones and tablet computers, based on the Linux operating system. It was developed by the Open Handset Alliance which is led by Google. CyanogenMod is a replacement firmware based on Android, developed to replace the official firmware distributed with smartphones and tablet computers. The CyanogenMod firmware provides access to core Android functionality that most phone providers hide from their users.

Table 5.2: Key-value pairs with the most instances across all projects

key	value	instances	distinct projects
`Priority`	`Medium`	66,671	981
`Type`	`Defect`	58,019	961
`Type`	`Enhancement`	17,255	570
`ReportedBy`	`User`	7,938	1
`Priority`	`Low`	6,537	466
`Priority`	`High`	5,228	482
`ReportedBy`	`Developer`	3,138	1
`Priority`	`Critical`	2,747	312
`Priority`	`Normal`	1,941	2
`OpSys`	`All`	1,798	131

Table 5.2 shows the ten most-used key-value pairs across all projects. In all, developers had used 2,042 different key-value pairs to annotate tasks. The two key-value pairs that Google suggests as default for new tasks, `Priority-Medium` and `Type-Defect`, were most commonly used. In addition, key-value pairs indicating by whom a task was reported and to what operating system it belonged were common.

It is interesting to note that some key-value pairs were used by almost all projects in the data, while others were only used by one or two projects. For example, the `ReportedBy` key was only used by one project (Android), and two projects used `Priority-Normal` instead of `Priority-Medium`.

Table 5.3: Key-value pairs with the most instances in Android

key	value	instances
`Priority`	`Medium`	18,516
`Type`	`Defect`	14,779
`ReportedBy`	`User`	7,938
`Type`	`Enhancement`	3,850
`ReportedBy`	`Developer`	3,138
`Version`	`2.2`	1,051
`Component`	`Applications`	890
`Component`	`Tools`	702
`Component`	`Framework`	684
`Version`	`2.3`	530

For tasks pertaining to the Android project, 91 different key-value pairs had been used. Table 5.3 shows the ten most-used ones. Compared to the most-used key-value pairs from all projects, Android developers put more emphasis on the components that the tasks belonged to, and they used key-value pairs to indicate versions.

Table 5.4: Key-value pairs with the most instances in CyanogenMod

key	value	instances
`Type`	`Defect`	3,088
`Priority`	`Normal`	1,939
`Priority`	`Low`	1,526
`Type`	`Enhancement`	611
`Model`	`D-S`	218
`Model`	`Nexus1`	202
`Version`	`7.0.3`	158
`Priority`	`Medium`	144
`Version`	`6.0.0`	110
`Version`	`7.0.0`	72

CyanogenMod developers used 154 different key-value pairs to manage their tasks. Table 5.4 shows the most-used ones. The CyanogenMod community used the key-value pair functionality to indicate which smartphone or tablet computer model a particular task belonged to.

The keys that were used most often in key-value pairs were also analyzed. Appendix C.1 shows the results for all projects, for Android, and for CyanogenMod.

Table 5.5: Labels with the most instances across all projects

label	instances	distinct projects
`Usability`	1,866	219
`Performance`	357	86
`Maintainability`	244	80
`Software`	147	1
`Security`	141	46
`CompilerBug`	140	1
`Escarpod`	127	1
`UI`	117	4
`Backlog`	105	2
`Bluetooth`	100	1

In total, the developers from all 1,000 projects used 524 different labels as part of their task management efforts. Table 5.5 shows the ten most-used ones. Similar to the findings presented in the previous chapter, labels were often used to communicate cross-cutting concerns, such as usability, performance, and maintainability. The next section presents a more detailed analysis of the different kinds of labels and key-value pairs that emerged from the data.

It is interesting to note that only one label – `Usability` – was used by more than 10% of the projects in the data. Four out of the ten labels in Table 5.5 were only used in a single project.

Table 5.6: Labels with the most instances in Android

label	instances
`SubcomponentOpenGL`	22
`SecurityProblem`	2
`Regression`	2

Developers of Android mainly used key-value pairs to manage their tasks. As shown in Table 5.6, only three labels had been used.

Compared to Android, developers of CyanogenMod used labels significantly more. In total, 51 different labels had been used to annotate tasks. Table 5.7 shows the ten most-used ones. Labels were mostly used to indicate different aspects of smartphones and tablet computers, such as Bluetooth, camera, or SMS.

Table 5.7: Labels with the most instances in CyanogenMod

label	instances
`Bluetooth`	100
`Phone`	97
`Camera`	87
`WiFi`	81
`i18n`	72
`CMExtras`	68
`SMS`	67
`3rdPartyApp`	63
`Audio`	57
`Usability`	56

RQ3: Characteristics of Labels and Key-Value Pairs

Through qualitative coding, codes were assigned to each of the labels and each of the keys in the data that accounted for the "heads" of the "long tails" in the distributions. The same coding scheme that was developed for the studies of tagging in IBM's Jazz was used (see Section 3.1).

Figure 5.4 shows the results from the classification of labels. The results were very similar to what we found for tags in IBM's Jazz (see Figure 3.6). Most of the labels indicated a particular component (e.g., `Browser` or `Installer`), or they were used to communicate planning-related concerns (e.g., `V1.2.0` or `Week2`). In addition, there were seven labels for cross-cutting concerns (e.g., `Usability` or `Performance`), and five labels for each of documentation (e.g., `Book` or `Examples`), environment (e.g., `3rdPartyApp` or `Symbian3`), and testing (e.g., `Patch` or `Tests`). Labels for architectural concerns and collaboration were less prevalent, and in this sample, labels were not used for idiosyncratic or tooling-related concerns.

The number of instances of a label being applied to a task was higher for cross-cutting labels than for any other kind. While there were not many label keywords indicating a cross-cutting concern, each one of the keywords had been applied to many tasks. Opposite to that, there were a lot of label keywords indicating a component, but each one of those keywords was not used very often.

The different keys that developers used as part of key-value pairs were also classified. Figure 5.5 shows the results. The classification is dominated by the two keys that had been used most often, and that are part of the default that Google suggests for new projects: `Priority` and `Type`. Most keys dealt with planning-related concerns

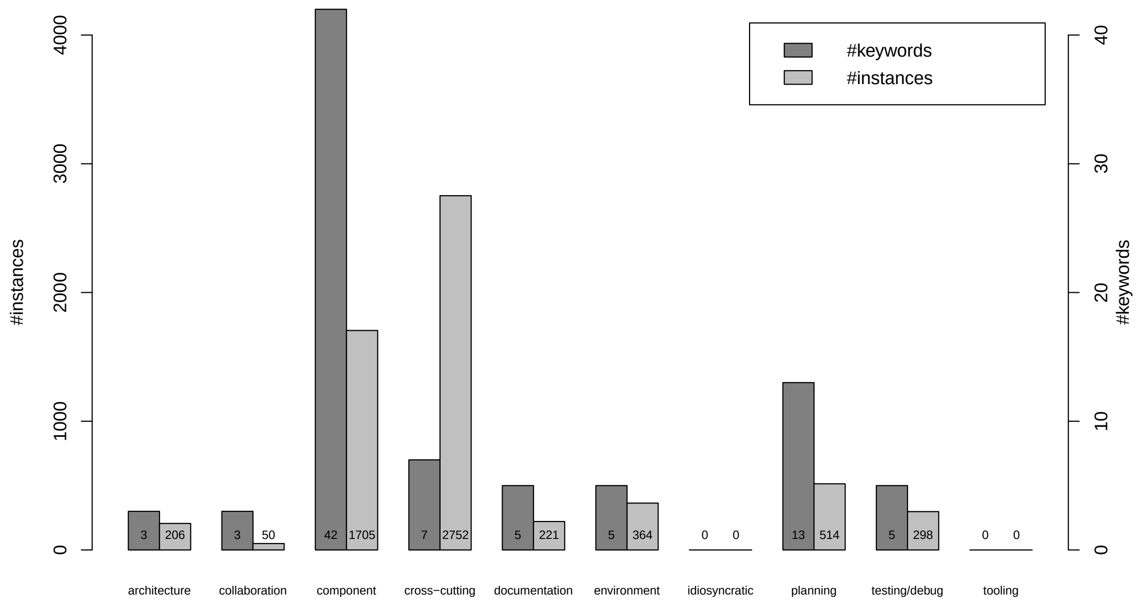


Figure 5.4: Number of label keywords and instances per category

(e.g., `Version` or `Release`), with components (e.g., `Subcomponent` or `Module`), or with the development environment (e.g., `Platform` or `Model`). In this sample, key-value pairs were not used to indicate tasks related to documentation or architecture.

RQ4: Surfacing of Key-Value Pairs

To understand which task properties different projects surface as part of their task management system, I analyzed which columns each of the 1,000 projects displayed by default. When setting up a project on Google Code, Google suggests the following seven columns by default: ID, Type, Status, Priority, Milestone, Owner, and Summary + Labels. For each project, an *edit distance* was calculated as the number of columns they added to the default plus the number of columns they removed from the default. Out of the 1,000 projects, 187 changed this default setting. Table 5.8 shows the edit distance for all projects.

The most common configurations were the default (813 projects), adding a Stars column (19 projects), deleting the Milestone column (13 projects), and adding a Component or a Reporter column (7 projects each). Table 5.9 shows the most displayed

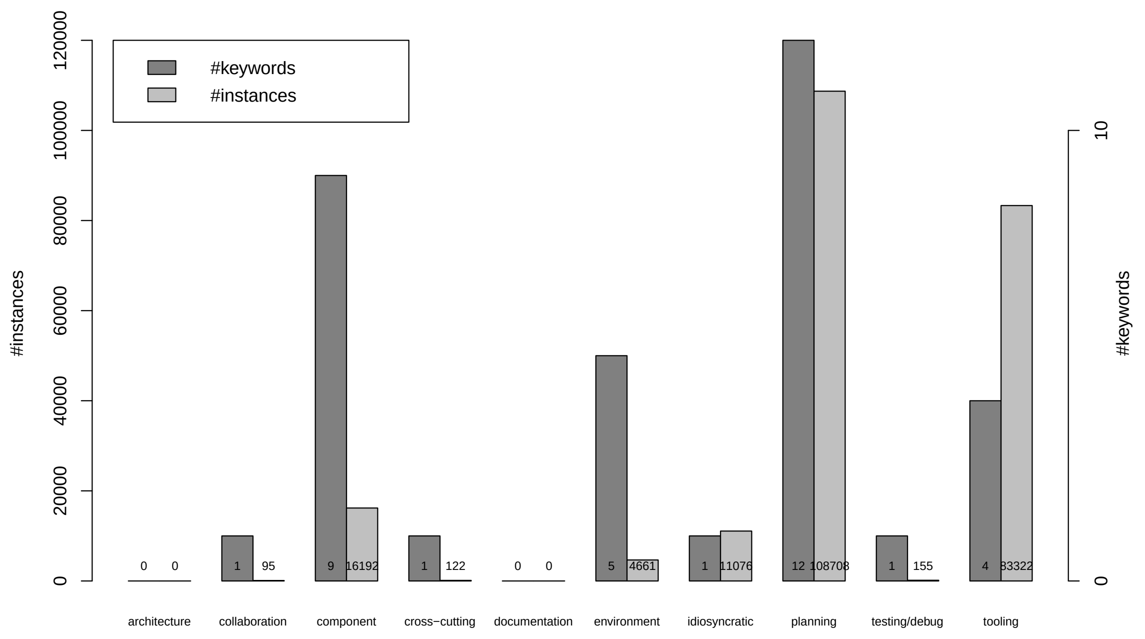


Figure 5.5: Number of keys and instances per category

columns. The first seven columns are part of the default, the remaining three are not.

5.1.4 Discussion

Interestingly, among the most-used labels, there were three different keywords that developers used to indicate that a particular task was easy to solve: `QuickTask`, `BiteSize`, and `Gentle`. For open source projects, such as the ones hosted on Google Code, it may be particularly important to indicate which tasks are easy to accomplish in order to attract new developers to a project.

The labelling feature was predominantly used to capture concerns related to software development tasks. Most of these concerns were either cross-cutting concerns that were captured using labels, or planning-related concerns which were captured using key-value pairs. An interesting case was the use of the `Affects` key by one of the projects to communicate cross-cutting concerns. The key had three different values: `Maintainability`, `Performance`, and `Usability`. In the sample, this was the only case where developers used key-value pairs rather than labels to indicate cross-cutting concerns.

Table 5.8: Edit distance of all projects

edits	projects
0	813
1	69
2	48
3	26
4	23
5	12
6	4
7	4
8	0
9	1

Table 5.9: Most displayed columns

column	projects
ID	996
Summary + Labels	995
Status	993
Type	990
Priority	978
Owner	954
Milestone	907
Stars	58
Component	35
Reporter	28

While the distinction into basic labels and key-value pairs offers more flexibility to the developers, most of the teams only took advantage of that flexibility to a certain degree. Some projects added an extra column, such as `Reporter` or `Component`, to their task management system, but most projects used the default setting suggested by Google Code. The basic labelling functionality may provide enough functionality to the developers to communicate all the important concerns.

5.1.5 Limitations

The first limitation of this study regarding labelling on Google Code lay in the sampling methodology. The 1,000 most popular and active projects were sampled according to Google's own ranking algorithm. The details of this algorithm are not public, and it is impossible to know exactly how the projects are ranked. In particular if labelling behaviour was part of Google's ranking algorithm, the validity of this sample would be limited. However, no claims were made based on the sampling methodology as only the labelling behaviour between different projects was compared. A random sample was not possible in this case because Google Code does not provide functionality to access all projects.

For the qualitative analysis, only the "heads" of the "long tails" in the distribution of labels and key-value pairs were coded. Although the remaining labels and keys may not have been used as frequently, the sampling method may have missed labels and keys that play important roles in the development processes of the 1,000 projects. However, a more fine-grained analysis is outside of the scope of this thesis.

Each project on Google Code can define labels and key-value pairs that will be set as default for all new tasks. These defaults can be adjusted based on the role of the developer creating the task (i.e., whether the task is created by a developer on the project, or by a developer from outside the project). Unfortunately, it is not possible to extract these default settings, and it is impossible to say to what extent the labels and key-value pairs observed were the result of these default settings. However, I argue that the labels and key-value pairs that are reported here were actually attached to the tasks and were used to manage them, whether they were there by default or not.

5.1.6 Summary

In this study on the role of social media artifacts – in particular labels and key-value pairs – in task management of software development teams using Google Code, I was able to confirm the findings from the previously described studies with development teams using IBM's Jazz. Tags or labels have been adopted by software developers to communicate concerns, and different kinds of tags and labels have emerged for processes that require meta data but are not formalized.

Google Code did not offer any awareness mechanisms such as the dashboards and feeds described in the previous chapter. Therefore, this discussion of labels and key-value pairs concludes the work on task management. In Part IV, the findings from these studies are used to build a model of social media artifacts in collaborative software development. Before that, Part III presents the findings on the role of social media artifacts in documentation.

Part III

Documentation

Chapter 6

Documentation of IBM's Jazz

Documentation plays an important role in many software organizations. Documentation can be disseminated to customers using a variety of media artifacts, including traditional help files, manuals, videos, and technical articles. In addition, social media artifacts, such as blogs and wikis, have emerged as important ways of documenting software in recent years. However, there is little advice on what channels are best suited to disseminate documentation.

This chapter presents a study of traditional and social documentation artifacts created by IBM's Jazz team. Using grounded theory, a model that characterizes documentation artifacts along several dimensions, such as content type, intended audience, feedback options, and review mechanisms, is developed. These findings lead to actionable advice for industry by articulating the benefits and possible shortcomings of the various communication channels in a documentation portal.

6.1 Documentation on the Community Portal jazz.net

Software development is knowledge-intensive [146], and the effective management and exchange of knowledge is key in every software organization. Knowledge can be distributed through various forms of documentation, from formal documentation and technical articles, to blogs and wikis. However, many organizations struggle with the effective exchange and dissemination of documentation across the community [98, 187].

Although various forms of documentation exist, software projects encounter many knowledge management challenges [62, 140]: How should documentation be distributed? How should documentation be kept up-to-date? How should feedback be solicited? How should documentation be organized for easy access? There is no roadmap for what kind of information is best presented in a given artifact, and new social media forms of documentation, such as wikis and blogs, have evolved [111]. Unlike more formal mechanisms, wikis and blogs are easy to create and maintain. However, they do not offer the same authoritativeness that comes with traditional documentation, and they can become outdated and less concise over time [42]. While the informality of a wiki page or blog is sometimes enough, users often expect reviewed technical articles.

One mechanism that has emerged recently and brings various communication channels together is the use of web or *community portals*. Such portals are not just used in software communities, but are essential to companies, such as Amazon, eBay or TripAdvisor, where they enable the development of communities around products. Similarly, many of today's software projects wish to solicit feedback and input from a broader community of users, beta-testers, and stakeholders. While several projects use community portals, such as Microsoft's MSDN or IBM's Jazz, the use of portals for software development projects has not been studied yet. It is unclear how portals can play an effective role in software development, and how different communication channels and artifacts should be utilized.

This chapter presents a study of a successful software project – the IBM Jazz client Rational Team Concert (RTC) – and its use of a community portal. The methodology follows a grounded theory approach, using data from 13 semi-structured interviews with developers, ethnographic field notes gathered during observations on-site, and quantitative data on the content of the community portal. Based on the findings, a model of artifacts in a community portal that characterizes the artifacts along different dimensions, such as intended audience, content type, feedback options, and review mechanisms, is developed. Actionable advice on how a community portal can provide benefits to a software project is provided, and the effective use of the different channels and artifacts available in a portal is discussed. The shortcomings of different artifacts are also identified, and improvements to current tools and processes are suggested.

The remainder of this section is structured as follows. Section 6.1.1 summarizes related work on knowledge management and documentation in software development, and the community portal that was studied is described in detail in Section 6.1.2. Section 6.1.3 describes the methodology, and the model of artifacts in a community portal is presented in Section 6.1.4. The findings are discussed in Section 6.1.5, and the limitations of the study are described in Section 6.1.6 before summarizing the contributions from this work in Section 6.1.7.

6.1.1 Related Work: Knowledge and Documentation in Software Development

Knowledge management has long been recognized as essential to software development [87]. Rowley [149] defines knowledge management as *"concerned with the exploitation and development of the knowledge assets of an organization with a view to furthering the organization's objectives. The knowledge to be managed includes both explicit, documented knowledge, and tacit, subjective knowledge."* How knowledge is managed in an organization depends on the particular style of software development. Plan-based or traditional methods usually rely on the management of explicit knowledge, while Agile methods rely on the management of tacit knowledge [128].

The number of studies on knowledge management in software development projects is limited. An overview of knowledge management in software development was conducted by Rus *et al.* [150]. The focus of their review is on motivations for knowledge management, different approaches to knowledge management, and factors that are important when implementing knowledge management strategies in companies.

Within knowledge management, the transfer of knowledge is particularly challenging. As described by Komi-Sirviö *et al.* [102], sharing knowledge is difficult in the light of short-term goals and companies often fall back to needs-based approaches for knowledge transfer. Knowledge transfer is essential when integrating new developers into a team. In a recent study, Dagenais *et al.* [41] identify early experimentation, internalization of structures and cultures, and progress validation as the main factors for the integration of newcomers. They also found that documentation was often not helpful and outdated.

Documentation has long been prominent in the list of recommended software engineering practices [47]. In 1986, Parnas and Clements [135] described a design process in which documentation plays a major role. They argue that many developers regard documentation as a necessary evil, written only as an afterthought to adhere to bureaucratic regulations. Therefore, documentation ends up being incomplete and inaccurate. Parnas and Clements identified four organizational issues that lead to those problems: poor organization of the documents which makes it harder to maintain them, boring prose that leads to inattentive reading, confusing and inconsistent terminology, and myopia caused by authors who know the documented system too well to take a comprehensive point of view.

In a survey of software professionals, Forward and Lethbridge [63] found that content, up-to-dateness, availability, and the use of examples are the most important attributes of documents. In a related study, Lethbridge *et al.* [109] explored how software engineers use and maintain documentation. They found that most software engineers do not update documentation in a timely manner, with the exception being highly-structured, easily-maintained forms of documentation, such as test cases and in-line comments.

Kajko-Mattsson [98] reports on a study in which she found poor documentation practices, even though her interviewees understood the necessity of documentation. Her participants identified the lack of a detailed documentation model as the main problem. Similar results were found in a study by Visconti and Cook [187]. They discovered a maturity gap between policies and the adherence to those policies.

Dagenais and Robillard [42] conducted a study to understand how documentation in open source projects is created and maintained. Among other findings, they report that the use of wikis for documentation has several shortcomings, such as a lack of authoritativeness, and that the use of a separate documentation team results in inconsistencies and out-of-date documentation.

Recent trends, such as agile documentation [8], suggest that executable products should be preferred over static documentation. From the agile point of view, documentation should be just good enough and it should only be updated "when it hurts" [8]. The rationale behind this is that the fundamental issue is communication, not documentation, and that comprehensive documentation does not ensure project success. While the developers in this study did not use agile documentation, only some of the documentation was formally organized, whereas other parts of the community portal evolved without strict rules.

As mentioned earlier, the use of community portals by software development projects has not been studied yet. Several open source projects, such as Eclipse[1], PHP[2] or the Apache HTTP Server[3], offer portals that bring together a variety of ar-

[1]http://www.eclipse.org/

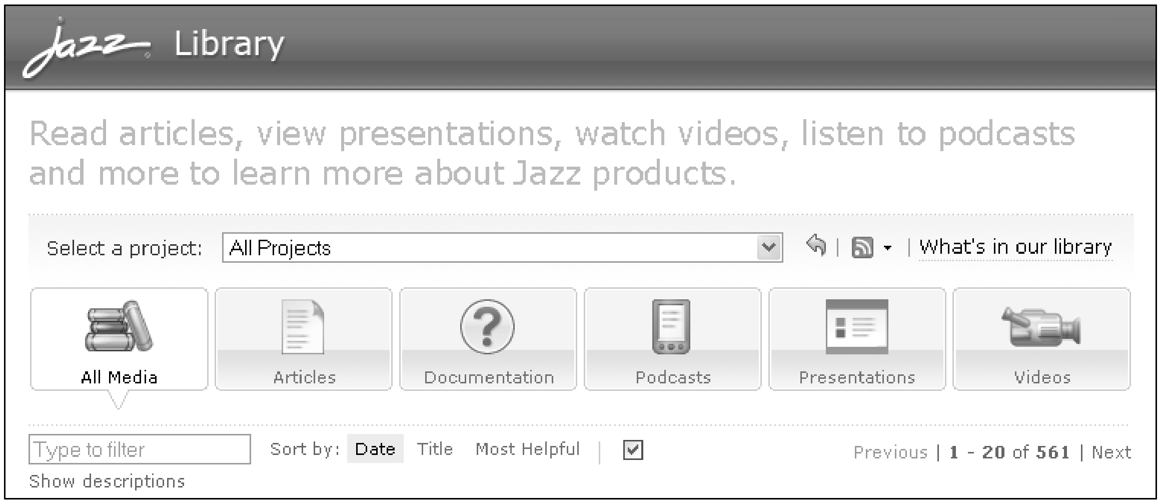
[2]http://www.php.net/

[3]http://httpd.apache.org/

tifacts, such as bug trackers, mailing lists, forums, wikis, and events. More recently, commercial products, such as Microsoft's Visual Studio or IBM's Jazz, have started using community portals. For example, the MSDN portal[4] features a library with API references, code samples and tutorials, a learning section, forums, and a blog. Next, the IBM Jazz community portal is described in more detail.

6.1.2 The Community Portal jazz.net

The jazz.net community portal hosts the entire development process including work items, developer mailing lists, the development wiki, forums, the team blog, and a library. The library features technical articles, podcasts, presentations, videos, and official product documentation, with tags organizing all artifacts in the library. All artifacts on jazz.net contain information that indicates their associated product, with some artifacts belonging to more than one product. Figure 6.1 shows a screenshot of one of the portal pages. All content is accessible to users after a free registration.



Read articles, view presentations, watch videos, listen to podcasts and more to learn more about Jazz products.

Select a project: All Projects

Figure 6.1: The IBM Jazz community portal

Technical articles are written by developers and product experts, and they explore tasks, use cases, solutions, and concepts in depth. Podcasts stem from various sources, including news and updates about Jazz products, and often feature Jazz developers and managers. The presentations have been given by developers and product managers at conferences, or they are simply informational. Videos are produced by Jazz

[4]http://msdn.microsoft.com/en-us/

developers or experts, and typically demonstrate the use of Jazz products or introduce enhancements. The official product documentation consists of HTML content that is part of the product release. The content is produced by a separate documentation team and is also accessible through the product help menus. Table 6.1 shows the different RTC-related artifacts that were available on jazz.net at the time of this study. Developers started to contribute to the community portal in the summer of 2007.

Table 6.1: RTC-related artifacts on jazz.net

type	amount
forum	8,413 threads
wiki	3,334 pages
mailing list	3,292 messages
blog	77 posts
official product documentation	4 manuals
library	93 articles
	58 videos
	30 presentations
	7 podcasts

Users of IBM's Jazz products and the jazz.net community portal are either end users or client developers. End users are typically software developers who use one or several of the Jazz products to develop their own products. Client developers are typically IBM employees who are in charge of developing extensions to Jazz or integrating other IBM products with Jazz.

6.1.3 Research Methodology

This section outlines the research methodology used to understand how software documentation is communicated through the Jazz portal. The study took place with the development team of RTC at IBM.

Data Collection

The methodology followed a mixed-methods approach, collecting quantitative and qualitative data to allow for triangulation. To gather quantitative data on jazz.net, web scraping was used to download the artifacts shown in Table 6.1, along with their tags. At the time of data extraction, the RTC team was working on milestones towards the 3.0 release. The quantitative data was collected to gain initial insights into the kinds of artifacts available on the community portal. These insights were used to guide the qualitative data collection that was done through a series of 13 semi-structured interviews. Seven interviews were conducted in person at an IBM location, and the remaining six interviews were conducted by phone. I interviewed five developers (**J-D1**, **J-D3**, **J-D8**, **J-D9**, **J-D0**), four component leads (**J-L1**, **J-L2**, **J-L3**, **J-L4**), the RTC development manager (**J-M**), the RTC project administrator (**J-A**), and two client developers (**J-C1**, **J-C2**) that worked on Jazz-related projects at IBM.

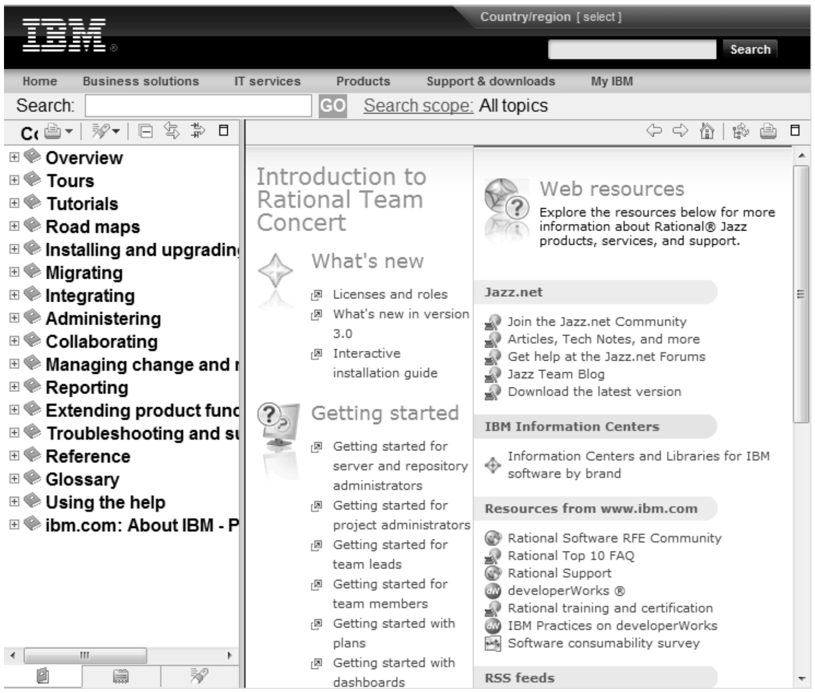
The initial interview script contained about 40 questions on participants' use of jazz.net. Appendix A.3 shows the set of questions used in the interviews. As expected with grounded theory, new questions emerged which led to refinements of the interview questions used during the course of the study. I asked the interviewees how they learned about Jazz functionalities, what kind of questions they asked using jazz.net, and how they used the published artifacts. The questions also investigated whether they had ever contributed content, and if so, who or what had triggered that contribution. I asked if there were perceived gaps in the documentation, and about potential tool and process enhancements. Follow-up questions were used for clarifications and additional details to understand the scenarios that had led to the creation and use of different artifacts. In addition, for this study, I spent three months on-site, frequently engaging in informal discussions with developers regarding their use of jazz.net, and facilitating the member checking of the findings. These observations were recorded using ethnographic field notes and allowed for insights into the internal processes that were not revealed in interviews, in particular the processes that would ultimately lead to the creation of new artifacts.

Data Analysis

The data analysis followed the grounded theory approach as described by Corbin and Strauss [37]. Grounded theory implies that data collection and analysis are interrelated processes, and that concepts are the basic units of analysis. These concepts are obtained using "open coding" in which the collected data is conceptualized line by line and concepts are only created when they are present repeatedly. Open coding was applied to the transcripts of the interviews, to the data downloaded from jazz.net, and to the field notes collected on-site. Based on the concepts, more abstract categories are developed and related. Each category is developed in terms of its properties, dimensions, conditions, and consequences. In the next step, called "axial coding", data is put together in new ways, thus making explicit connections between categories and sub-categories. Sampling in grounded theory is done on theoretical grounds where incidents and events are sampled rather than subjects or data sources. In the final step of "selective coding", the core category is identified and systematically related to other categories. All quantitative and qualitative data collected in this study was considered during the analysis [37].

Unlike the studies described in Part II, I did not have specific research questions at the beginning of this study. In grounded theory, the core category only becomes clear during open coding. Since researchers do not start with a problem statement but an interest in the field, it is impossible to derive research questions up-front [73].

6.1.4 Findings



Introduction to Rational Team Concert

What's new

- Licenses and roles
- What's new in version 3.0
- Interactive installation guide

Getting started

- Getting started for server and repository administrators
- Getting started for project administrators
- Getting started for team leads
- Getting started for team members
- Getting started with plans
- Getting started with dashboards

Web resources

Explore the resources below for more information about Rational® Jazz products, services, and support.

Jazz.net

- Join the Jazz.net Community
- Articles, Tech Notes, and more
- Get help at the Jazz.net Forums
- Jazz Team Blog
- Download the latest version

IBM Information Centers

- Information Centers and Libraries for IBM software by brand

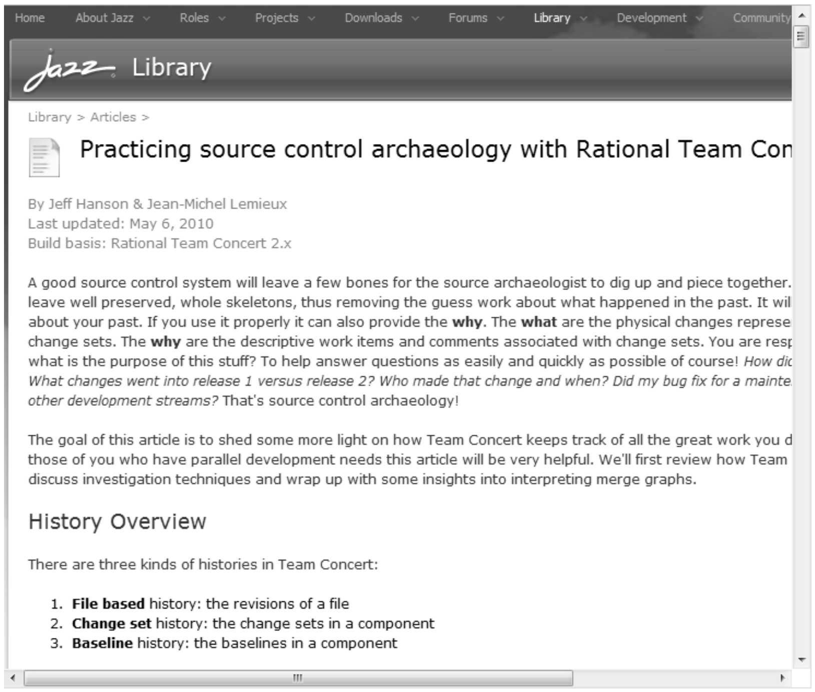
Resources from www.ibm.com

- Rational Software RFE Community
- Rational Top 10 FAQ
- Rational Support
- developerWorks ®
- Rational training and certification
- IBM Practices on developerWorks
- Software consumability survey

RSS feeds

Figure 6.2: Screenshot of the official product documentation

The "core category" identified in this grounded theory study is a set of key characteristics of different artifacts in a community portal. The artifacts can be distin-



Practicing source control archaeology with Rational Team Con

By Jeff Hanson & Jean-Michel Lemieux
Last updated: May 6, 2010
Build basis: Rational Team Concert 2.x

A good source control system will leave a few bones for the source archaeologist to dig up and piece together.
leave well preserved, whole skeletons, thus removing the guess work about what happened in the past. It wil
about your past. If you use it properly it can also provide the **why**. The **what** are the physical changes represe
change sets. The **why** are the descriptive work items and comments associated with change sets. You are resp
what is the purpose of this stuff? To help answer questions as easily and quickly as possible of course! *How dic*
What changes went into release 1 versus release 2? Who made that change and when? Did my bug fix for a mainte
other development streams? That's source control archaeology!

The goal of this article is to shed some more light on how Team Concert keeps track of all the great work you d
those of you who have parallel development needs this article will be very helpful. We'll first review how Team
discuss investigation techniques and wrap up with some insights into interpreting merge graphs.

History Overview

There are three kinds of histories in Team Concert:

1. **File based** history: the revisions of a file
2. **Change set** history: the change sets in a component
3. **Baseline** history: the baselines in a component

Figure 6.3: Screenshot of a technical article

guished along eight different dimensions that emerged from the axial and selective coding as part of the data analysis. The dimensions underline the particular role of each artifact.

Content: the type of content typically presented in the artifact.

Audience: the audience for which the artifact is intended.

Trigger: the motivation that triggers the creation of a new artifact.

Collaboration: the extent of collaboration during the creation of a new artifact.

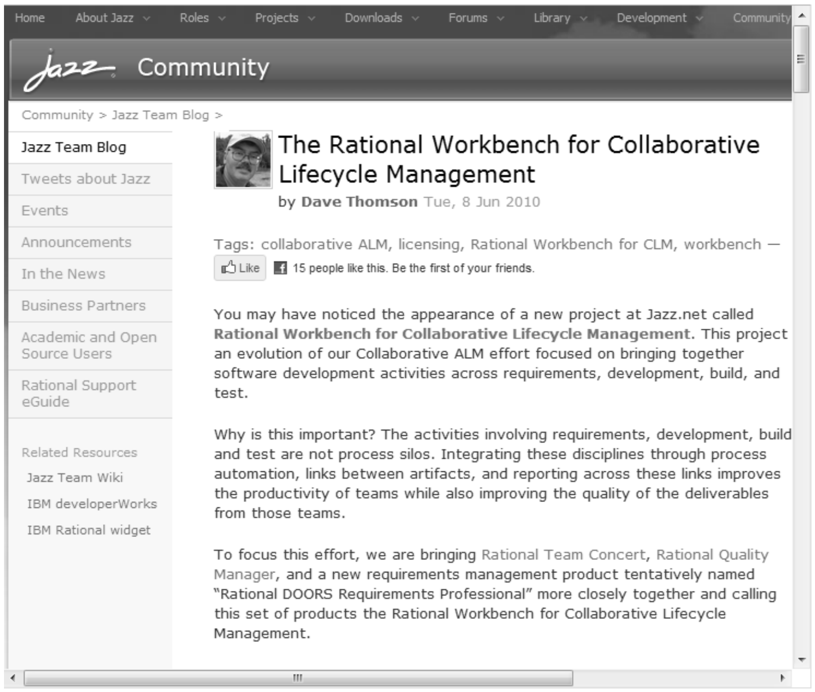
Review: the extent to which new artifacts are reviewed before publication.

Feedback: the extent to which readers can give feedback.

Fanfare: the amount of fanfare or "buzz" with which a new artifact is released.

Time Sensitivity: the time sensitivity of information in the artifact.

For the official product documentation, technical articles, blog posts, and the developer wiki, these dimensions were analyzed in detail. I focus on these particular



The Rational Workbench for Collaborative Lifecycle Management

by Dave Thomson Tue, 8 Jun 2010

Tags: collaborative ALM, licensing, Rational Workbench for CLM, workbench —

15 people like this. Be the first of your friends.

You may have noticed the appearance of a new project at Jazz.net called **Rational Workbench for Collaborative Lifecycle Management.** This project an evolution of our Collaborative ALM effort focused on bringing together software development activities across requirements, development, build, and test.

Why is this important? The activities involving requirements, development, build and test are not process silos. Integrating these disciplines through process automation, links between artifacts, and reporting across these links improves the productivity of teams while also improving the quality of the deliverables from those teams.

To focus this effort, we are bringing Rational Team Concert, Rational Quality Manager, and a new requirements management product tentatively named "Rational DOORS Requirements Professional" more closely together and calling this set of products the Rational Workbench for Collaborative Lifecycle Management.

Figure 6.4: Screenshot of a blog post

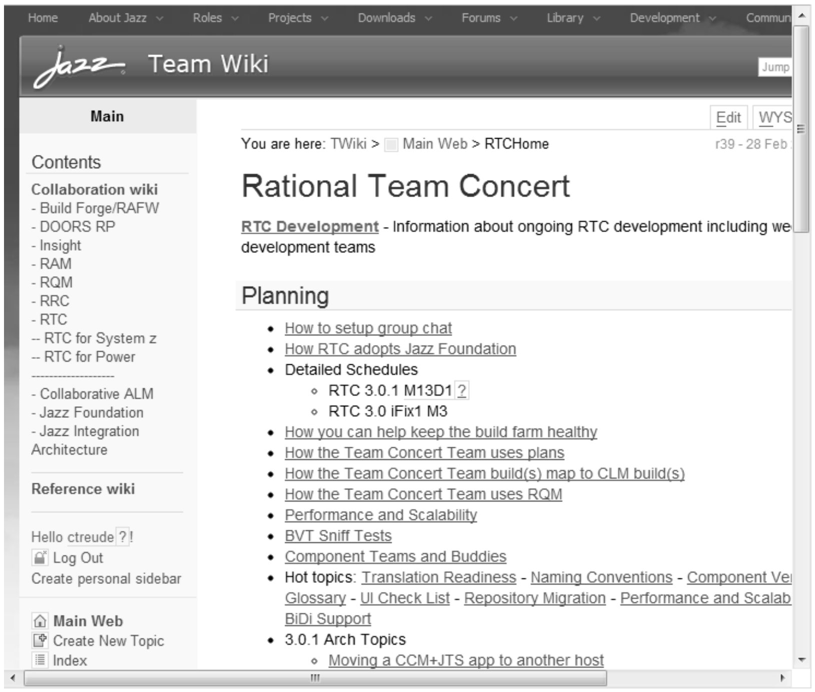
kinds of artifacts because they are also common in other community portals and in software development in general. Figures 6.2 through 6.5 show examples for each artifact, and Tables 6.3 through 6.6 summarize the findings for each kind of artifact. The following sections report on each of the dimensions in detail. In addition, several explicit and implicit connections between the artifacts are highlighted.

Table 6.2: Most-used tags in jazz.net library by type

type	top two tags
article	`agile` (15), `getting started` (15)
video	`how-to` (12), `introduction` (7)
presentation	`RSC 2009` (9), `agile` (5)
podcast	`agile` (3), `scrum` (2)
official doc	`help` (4)

What Content is Communicated?

Content is the first dimension along which the different artifacts of the community portal can be distinguished. The official product documentation is based on features and does not cover scenarios: *"I would try to cover all the corner cases, and I found*



Main

Contents

Collaboration wiki
- Build Forge/RAFW
- DOORS RP
- Insight
- RAM
- RQM
- RRC
- RTC
-- RTC for System z
-- RTC for Power

- Collaborative ALM
- Jazz Foundation
- Jazz Integration Architecture

Reference wiki

Hello ctreude ?!
Log Out
Create personal sidebar

Main Web
Create New Topic
Index

You are here: TWiki > Main Web > RTCHome r39 - 28 Feb

Rational Team Concert

RTC Development - Information about ongoing RTC development including we development teams

Planning

- How to setup group chat
- How RTC adopts Jazz Foundation
- Detailed Schedules
  - RTC 3.0.1 M13D1 ?
  - RTC 3.0 iFix1 M3
- How you can help keep the build farm healthy
- How the Team Concert Team uses plans
- How the Team Concert Team build(s) map to CLM build(s)
- How the Team Concert Team uses RQM
- Performance and Scalability
- BVT Sniff Tests
- Component Teams and Buddies
- Hot topics: Translation Readiness - Naming Conventions - Component Ve Glossary - UI Check List - Repository Migration - Performance and Scalab BiDi Support
- 3.0.1 Arch Topics
  - Moving a CCM+JTS app to another host

Figure 6.5: Screenshot of a wiki page

those were often omitted in the help doc because the effort to get into those corner cases were lengthy things. You would have to have a whole section of how to put yourself into a corner, and then a whole other section of how to get yourself out of this corner." (J-D3) However, documenting scenarios is important: *"It's always good to document a widget, but it's more important in many cases to document a process and being able to follow the process of how you do an upgrade or how you do a plan and all the widgets that you touch during that. [...] It's the context of how you use the widget that's much more important."* (J-C2)

This need is addressed by technical articles that feature scenarios and offer more depth than the official documentation. J-D0 described the reason to write an article: *"Because I wanted to put in a lot of screenshots, and also explain what's happening. [...] You had to write some text around it. Why you're doing something and why is that needed."* Technical articles also differ in their style of writing: *"Articles spice it up a little. There's more pictures and [they are] more personal I guess. People have their own style."* (J-D8)

Tags also shed light on the content of different kinds of artifacts. The tags are assigned by a single person (J-L3). Table 6.2 shows the most-used tags per type. Videos are typically used for high-level overviews or demos (J-L3, J-D9), whereas

Table 6.3: Role of the official product documentation

Official product documentation
*What **content** is communicated?*
feature descriptions (J-D3, J-C2)
no scenarios (J-D3, J-C2)
*Which **audience** is reached?*
buying customers (J-M)
*What **triggers** the production of new artifacts?*
a new product release
*To what extent is content produced **collaboratively**?*
content is produced by a separate documentation team (J-M, J-A, J-D1)
*To what extent is content **reviewed** before publication?*
content undergoes rigorous reviews (J-M)
*To what extent can readers give **feedback**?*
no option to give feedback
*With how much **fanfare** are new artifacts released?*
same fanfare as product releases (J-D3)
*How **time sensitive** is the information?*
information is outdated quickly (J-A, J-D3)
content is never updated

presentations are related to conferences.

Blogs add a personal note to the artifacts on the community portal: *"a personal view on something and not really documentation. [...] You want to make people aware of something or tell them about something, but more like a teaser, you can go somewhere else to find the actual documentation. It's more like you write about something new and cool and what you think about it."* (J-L2) Therefore, the blog also plays a role in marketing (J-L4, J-D1, J-C1). The most commonly-used tags on blog posts are `rational-team-concert` (26 posts), `video` (13 posts), `self-hosting` (12 posts), and `conference` (11 posts). Most blog posts are related to RTC and provide a conference report or give a view on the Jazz team's self-hosting experience.

The developer wiki plays many roles in the internal development processes, ranging from planning to descriptions of scenarios, detailed instructions or references to other resources. Occasionally, wiki content goes beyond supporting development processes: *"He's written some really good wiki topics. The material there is really going beyond what we typically have in the wiki and really should be as help or as articles."* (J-L1)

Table 6.4: Role of technical articles

Technical articles
*What **content** is communicated?*
scenarios, how-to (J-D3, J-D0)
typically more depth than the official product documentation (J-M, J-A, J-D3)
*Which **audience** is reached?*
individuals outside the development team (J-L1, J-L2, J-D3, J-D9, J-D0)
*What **triggers** the production of new artifacts?*
user questions (J-M, J-A, J-L1, J-L2, J-D1, J-D3)
organized documentation efforts (J-A, J-L2, J-L3, J-D1, J-D3, J-D8, J-D0)
feature promotion (J-L2, J-D1, J-D8)
*To what extent is content produced **collaboratively**?*
content is mostly produced through solo efforts (J-M, J-L2, J-D8, J-D0)
*To what extent is content **reviewed** before publication?*
content is reviewed before publication (J-L1, J-D1, J-D3, J-D8)
content is less formal than the official product documentation (J-M, J-D8)
*To what extent can readers give **feedback**?*
new feedback section recently implemented (J-L3)
*With how much **fanfare** are new artifacts released?*
new content is released without much fanfare (J-M, J-L1, J-L4)
some individuals learn about new content through twitter etc. (J-D8, J-C1)
*How **time sensitive** is the information?*
information is not outdated quickly (J-L2)
most information is related to one particular release (J-L1, J-L3, J-D3)
content is rarely updated (J-A, J-L2, J-D1, J-D8, J-D0)
producing new content takes time (J-D1, J-D3)

Which Audience is Reached?

A community portal caters to a diverse audience: *"A good percentage of our community are internal IBMers, business partners, students, academics, as well as customers."* (J-L3) From a documentation point of view, the distinction of end users and client developers is particularly interesting: *"There's two kinds of customers. We have customers who are using the product and trying to do something, and then we have third party developers. And third party developers are – for them to see the wiki makes sense, because we have instructions on how to make stuff."* (J-D1) In addition, some communication with customers is done outside of the community portal for confidentiality reasons (J-L1).

Table 6.5: Role of blogs

Blogs
*What **content** is communicated?*
case studies (J-C1)
personal views (J-L2, J-L3, J-C1)
marketing (J-L4, J-D1, J-C1)
*Which **audience** is reached?*
the community surrounding the project (J-L3, J-L4)
*What **triggers** the production of new artifacts?*
user questions (J-M, J-A, J-L1, J-L2, J-D1, J-D3)
feature promotion (J-L2, J-D1, J-D8)
*To what extent is content produced **collaboratively**?*
content is mostly produced through solo efforts (J-L4)
*To what extent is content **reviewed** before publication?*
content is reviewed before publication (J-L4)
*To what extent can readers give **feedback**?*
comment section exists
*With how much **fanfare** are new artifacts released?*
more fanfare than new articles (J-L4)
*How **time sensitive** is the information?*
information is time sensitive (J-A)
content is rarely updated (J-A)
producing new content takes time (J-L4)

The intended audience of the official product documentation, technical articles, and the blog is outside of the development team. The intended audience for wiki pages is more difficult to determine, and some of the developers have contradicting views on the role of the wiki: *"For the wiki, it's developers [...] that I actually have on my [chat] list or where I can put a name to a face."* (J-D3) However, customers are occasionally pointed to wiki pages: *"We also point customers to wiki pages. But wiki pages are typically [more] interesting for development teams. It's not as polished, usually, as articles and probably also not as maintained."* (J-L2) For customers, it is important to distinguish between official and unofficial wiki content: *"I'm not sure if customers should be taking that [wiki page] and making plans, planning their day on that."* (J-D3) However, outdated wiki content is of concern: *"Once or twice I've had customers say I'm trying to do stuff from the design document. And it's structured as instructions which is dangerous because they're not correct."* (J-D1)

Table 6.6: Role of wikis

Wikis
*What **content** is communicated?*
plans (J-D1, J-D3), scenarios (J-M, J-L2)
instructions (J-D3, J-C1), references (J-L1, J-D3)
*Which **audience** is reached?*
inside the development team (J-L1, J-L2, J-L3, J-L4, J-D0, J-C2)
outside the development team (J-M, J-A, J-L1, J-L2, J-D1, J-D3, J-D8)
*What **triggers** the production of new artifacts?*
need to support development work (J-M, J-L4, J-D3, J-D8)
*To what extent is content produced **collaboratively**?*
content is mostly produced through solo efforts (J-M, J-A, J-L1)
some content is produced collaboratively (J-L2, J-L4, J-D1)
*To what extent is content **reviewed** before publication?*
content is rarely reviewed (J-M, J-L1)
content is not formal (J-M, J-D1, J-D9)
*To what extent can readers give **feedback**?*
no option to give feedback
*With how much **fanfare** are new artifacts released?*
no fanfare for new content (J-D9)
*How **time sensitive** is the information?*
content is changed often (J-A, J-L1, J-D3, J-D9)
content can become stale (J-M, J-A, J-L1, J-L2, J-D1, J-D8, J-D9)
content is easy to add to (J-L1, J-D1, J-D8)

What Triggers the Production of New Artifacts?

There are no formal conventions as to what content should be externalized in technical articles, blog posts, or wiki pages: *"We haven't gotten to the point where we've had to dictate how often certain articles are developed. The content has flown fairly actively, especially with videos. People are getting [into] the habit of delivering release videos at each release with demos and how-to articles. So so far it's been able to happen organically."* (J-L3) Articles are not written systematically but on an *ad hoc* basis (J-D1, J-D0), and without a formal process to trigger articles on certain topics. Choosing the right things to document is subjective: *"The features we choose to work on, they're chosen in an* ad hoc *manner and we do the same thing with our documentation."* (J-D1) Initially, contributions to the blog were organized: *"When we first started doing it, a [...] bunch of us were pulled together and were asked to contribute."* (J-L4)

However, blog posts now occur in an *ad hoc* manner as well, and over the last three years, there have been 43 different blog authors.

The following paragraphs discuss common motivations for producing content that emerged from this study: user questions, organized efforts, self-promotion, and support of development work.

Content is externalized in the form of articles and blog posts when users repeatedly ask the same question (J-M, J-A, J-L1, J-L2, J-D1, J-D3), or when developers expect many questions on a subject: *"Because we have a feeling that there will be so many users that have the same questions that it will be more beneficial to sit down, write the article once."* (J-D3) Questions arise from the forum (J-A), they are reported back from customer representatives (J-L3), they come from feedback to other parts of the documentation (J-D0), or they come by email: *"I try and take every email conversation that I end up having and put it into an article."* (J-A) The motivation for creating wiki pages is similar: *"I wanted not to be the sole source of this information. I wanted them to have a place that they could go and look and refer back to."* (J-D9)

Some articles are the result of organized efforts among developers: *"There are efforts internally where groups of people get together and say, what's missing? [...] What are the customers asking for? We need this article or that article. [...] Some of the articles come up that way, and then other articles just happen organically as the team decides [...] they want to write an article on a particular topic."* (J-L3) Several interviewees reported that they had prompted articles from other individuals (J-A, J-L3) or that articles had been delegated to them (J-L2, J-D1, J-D3, J-D8, J-D0).

Technical articles and blog posts are also written by developers who want to promote a feature they authored (J-L2, J-D1) or who want to promote themselves: *"Part of my thought is also, when I put my name on something, if and when I ever want to be interviewed for something, I want it to be on my resume."* (J-D1) J-D8 had a similar motivation for his article: *"'Cause I wrote it, I want to get exposure."*

The externalization of content to the wiki mainly happens to support development work, either for an initial *"brain dump"* (J-D3), as a central entity for resources (J-D8), to facilitate a discussion (J-M, J-D3), or to *"stage something that needs to be formalized"* (J-D3). J-L4 explains: *"It becomes kind of an open email if you will, to evolve an idea."*

To What Extent is Content Produced Collaboratively?

Artifacts on a community portal also differ in the amount of collaboration that happens around them. The official product documentation is written by a separate documentation team, a process which involves a certain amount of interaction with the core developers (J-D3). Authoring technical articles, blog posts, and wiki pages is usually a solo effort, and only involves other individuals for reviews. For selected wiki pages, such as descriptions of parts of the system architecture, the amount of collaboration can be higher: *"There's one wiki page that we have on how to set up the development environment [...] and this is something where anybody can add information, and actually also does."* (J-L2)

The author information on articles is another indicator of collaborative content. Out of 93 technical articles, 35 have 1 author associated with them, 9 articles have 2 authors, 2 articles have 3 authors, and 1 article has 5 authors. An additional 34 articles have at least 1 team name as author, and 12 articles do not have any author information.

To What Extent is Content Reviewed Before Publication?

Since the official product documentation is part of the product release, it has to undergo a formal review process in which content is reviewed by technical writers as well as by the product teams of the documented components. *"They go through a pretty rigorous process of producing that content, working with the various teams that provide the features. [...] So the quality of it is very good."* (J-M) The official product documentation has commitment from the developers and it is the only artifact that is translated into other languages. Such a process is not transferable to all kinds of artifacts as it requires a lot of resources and therefore only happens once for every major release.

Technical articles are less formal. While there are internal reviews before content is posted online, the style is more personal: *"The articles tend to be an easy read, they're more down to earth and personal and not like you learn at school, how they have to be formal and very professional and impersonal."* (J-D8) Blog posts have to undergo a similar internal review process where one individual on the team proof-reads content before it is posted online. Initially, there was some guidance on blogging: *"When we first started writing blogs for jazz.net, someone who had been a blogger [...] before that, he basically gave us guidance in the Jazz sphere. He said read these guidelines* *before you write blogs."* (J-L4)

Wikis are the least formal of the artifacts and are rarely reviewed. However, contributors are aware that the wiki is open to the public and some of the wiki content, such as details of new features, has to be coordinated with management (J-M). The lack of review often leads to inconsistencies on the wiki: *"There [are] pages that I think they're just sitting there, there [are] just certain people who know about that – they're just often in their own little world."* (J-D9)

To What Extent Can Readers Give Feedback?

Feedback is another dimension that distinguishes the different kinds of artifacts on a community portal. While the official documentation does not allow for feedback, and the developer wiki is read-only for everybody outside the core developer team, a feedback section has recently been implemented for technical articles, podcasts, and other items in the library. 40 of the 58 videos on jazz.net are hosted on YouTube, which provides view counts (considered as implicit feedback) and allows comments. At the time of this study, the videos had a median of 1,074 views, ranging from 188 to 17,157[5]. Only 7 videos had comments, with a maximum of 2 comments per video.

Blog posts invite interaction with a commenting feature. At the time of this study, there was a total of 183 comments on 77 blog posts. 28 blog posts had no comments, 15 blog posts had 1 comment, and 7 posts had 2 comments. 27 posts had 3 comments or more, with a maximum of 12 comments per post. Blog posts with a lot of comments are typically product announcements or discussions of new functionality.

Community members can vote on blog posts. At the time of this study, 446 votes had been cast. The most voted-for post had 22 votes, and only 4 out of the 77 posts had not received any votes. On average, articles received 4.6 stars out of 5, and only 6 articles received less than 4 stars. A recent addition is the option to "like" posts using Facebook Connect, and 4 out of the 10 newest posts had already received more than 10 likes at the time of this study. While the web traffic[6] for artifacts, such as technical articles in the library, is monitored as well, pure numbers do not allow conclusions about the usefulness of the content (J-L3).

Feedback is often generic: *"I got a thumbs up, like good thing to do, but not any specific critique on my writing. More like, that's a good thing to do, keep doing that."*

[5]An introductory video for RTC, available online at http://www.youtube.com/watch?v=ILvsGQQqAF0.

[6]Data on web traffic was not made available.

(J-D9) Some developers even go out of their way to search for feedback: *"There's no feedback and I don't really know what's happened, I don't know if people read [the article]. [...] Then by searching just to see how exposed it was, I found a few forum discussions on jazz.net [about the article]."* (J-D8)

While the forum and the article comment feature were designed for users to give feedback (J-L3, J-D0), different feedback comes through different channels (J-L2, J-L4, J-D1). Questions about content on jazz.net are mostly asked through direct communication channels, such as chat (J-L2, J-D3) or email (J-A, J-L1). The customers often work within IBM and therefore have access to the internal instant messaging system (J-L2, J-C1).

Developers are aware that putting their name on artifacts can lead to many questions (J-D1) or spam (J-D0). *"Usually once you have documented something that people are using, then they come back to you directly with questions."* (J-L2) During the ethnographic observations, I encountered an instance of a developer deciding not to write an article because they anticipated a high workload due to questions and feedback.

With How Much Fanfare are New Artifacts Released?

Artifacts also differ in the fanfare with which they are released and in the number of people that are aware of a new artifact after its creation. While the official product documentation is released alongside the products once a year, blog posts have a higher impact than articles. *"Articles are a little bit quieter."* (J-L4) This is confirmed by developers who are not aware of new articles (J-M) or only read articles when prompted (J-L1). Other developers follow the @JazzDotNet Twitter account to learn about new articles (J-C1), or they check the feed on the front page of the portal (J-D8). New or changed wiki pages have the lowest amount of fanfare as the wiki implementation used by this group does not offer notifications.

How Time Sensitive is the Information?

Some artifacts are more time sensitive than others. The official product documentation becomes outdated quickly, mainly because it is only updated once a year. Technical articles are less time sensitive due to the nature of the topics: *"I think the topics are usually chosen so that they stay valid for a longer time. [...] An article that describes, for example, customization of a work item, these are longer-lived features* *that we do not break as easily."* (J-L2) For articles, knowing which release they belong to is important: *"We try to mark every article as relevant to a particular release."* (J-L3)

For articles and blog posts, the writing process takes a few days (J-L4, J-D1, J-D3), and there is more flexibility on when they are published: *"Sometimes we write articles after the fact. So we didn't get time, we delivered the feature late or something, or we didn't get to document it in time for the release, so then we say okay, we're going to write an article."* (J-M) Developers also choose to write articles after the code freeze point at the end of a release cycle[7] (J-D1). However, the review process can lead to publishing delays: *"I've had experiences in the past where I want to get something out in a timely way but because of the review it doesn't necessarily get there."* (J-L4)

Technical articles, blog posts, and wiki pages contain tools to update content, but most authors in this study have never updated their articles (J-L2, J-D1, J-D8). Since the website is hosted by a specific team, developers were unsure what the article change process would be after the initial publication (J-D8). In other cases, developers passed on the ownership of an article before they switched teams to make sure updates would happen: *"For the second release of the product, one of my team members here updated the article. [...] He took ownership."* (J-D0) Blog posts are more time sensitive than articles, because they express personal views and usually do not feature technical details (J-L2, J-L3, J-C1). However, only one of the developers in this study had made updates to a blog post (J-A).

In contrast, wiki pages are quick and easy to create and modify: *"I think the wiki is good 'cause it can be sort of fast and loose for getting content out there and the articles tend to be more reviewed, and they're also harder to change after the fact."* (J-L1) There is no formal process outlining when wiki pages should change, and some developers only update their wiki pages if they are reminded to do so by somebody else (J-A, J-D1) or if they happen to see that something is wrong (J-D8). Updates are usually done to describe new features (J-A), to answer questions (J-L1), or to reorganize a few pages (J-L1). Wiki pages aimed at customers are kept current (J-L1) and the index is also kept up-to-date (J-A).

Readers cannot expect all wiki pages to be up-to-date: *"If it's incorrect I would*

[7]A code freeze is a point in time in a software development process after which no changes are permitted to the source code. Code freezes are usually implemented to ensure that parts of the program that are known to work correctly will continue to do so.

say, it's a wiki, you can't expect that to be correct." (J-D3) A wiki page implicitly conveys the uncertainty of its content: *"It helps us to communicate to customers that it's actually not done yet."* (J-M) As observed by many interviewees, wiki pages can have stale content: *"We write a topic on something and then we walk away from it, we just ignore it."* (J-D1)

Artifact Connections

Artifacts on a community portal, with all their characteristics discussed in the previous paragraphs, cannot be treated separately. There are several explicit and implicit connections between them. For example, content or structure can be reused from different artifacts. The structure for an article can come from a wiki page (J-D3), a blog post can be distilled from wiki pages (J-L4), wiki pages can draw content from forum posts (J-L1), and forum posts can be referenced in work items (J-D8).

Several wiki pages have the potential to be turned into technical articles or blog posts, but that does not happen often: *"If we had something that we were formulating on the wiki and we're getting feedback and so forth and we're altering it over time – once we have decided that it's final, we could basically copy and paste all the stuff and put it in a better looking web page. An article possibly, depending on what it is, or some official document. I can't actually think of specific cases where it has [happened]. [...] We tend to leave a lot of the stuff [on the wiki]."* (J-M) Another example is an article created by J-L4: *"[I added] an article that basically has no content just a description and a link to our wiki [...] using all the keywords that make sense to find it. And that way when somebody goes in looking for this information, [...] we point them into the wiki, which isn't ideal but it does surface the content."*

Relationships between artifacts also exist from a client developer's perspective. J-C2 describes using the wiki and technical articles as his first place to go for questions, and posting to the forum if he cannot locate an answer: *"That's the fallback. And if I don't get a response in the forum I actually will post a defect or an enhancement."*

6.1.5 Discussion

This section presents advice on the findings from this case study of a successful software project that leverages a community portal.

Make Content Available, but Clearly Distinguish Different Media Forms

One major advantage of a community portal is the accessibility of all artifacts: *"Because it's actually just as likely that an answer for a question is going to be in a forum or in a mailing list as is it going to be in a wiki."* (J-C2) Even though not all content is produced with the intent to contribute to documentation, it can help developers inside and outside of the team to understand a system.

However, wiki pages that are created to support development work and that are often not as well structured as other documentation sources on the community portal should be flagged as draft material. This distinction is currently not always clear: *"So people do have confusion about what's real and what's – what's been approved versus just draft."* (J-A) Individuals from outside the core development team find it difficult to navigate the wiki (J-C2) and they encounter stale content (J-C1). The lack of structure is also a concern for developers on the team (J-D1, J-D9).

Clearly distinguishing different media forms is a challenge for search interfaces: *"If you do a search, you're confronted with a blend of all the media and different types of it."* (J-C1) Jazz.net offers a main search interface that searches the entire content of the community portal, as well as specific searches for the wiki, the library, and the forum. Most interviewees found the current search to be insufficient (J-L1, J-L2, J-L3, J-D1, J-D9, J-D0, J-C1, J-C2). As an alternative to a search, the wiki index page is kept up-to-date (J-L1) and tags are used on artifacts in the library.

Make it Easier to Move Content into More Formal Media Formats

Different media forms in community portals are often disconnected. Content is rarely transferred from wiki pages into more formal articles or into the official product documentation. Instead, developers create workarounds, such as articles, that point to a wiki page (J-L4). Often, good content is never published beyond wiki pages because developers lack the time to push for a new article or blog post (J-L1). Time is wasted by replicating information that already exists in other parts of the community portal (J-L4). Part of the problem is the use of a separate documentation team for producing the official product documentation. With the plethora of information on a community portal, the first step when writing a new piece of documentation should be to check if related content already exists.

Be Aware of the Implications of Different Media Artifacts and Channels

As shown with the model of documentation artifacts in a community portal, different media artifacts and channels have different implications (see Tables 6.3 through 6.6). To ensure the effective exchange and dissemination of content, developers and managers need to be aware of these implications:

- Content in wiki pages often becomes stale. Therefore, readers will not look at the wiki for reliable information, but rather use it as a backup option if information is not available elsewhere. To communicate important information to the community, articles and blog posts are better suited.
- The official product documentation is reviewed rigorously. With that in mind, it can serve as the most reliable way to communicate content in a community portal.
- When content is produced by a separate documentation team, updates may not be feasible. In such a case, information may become outdated quickly and can only be fixed with a new product release. Consequently, an update process for articles may be needed.
- In this project, new blog posts created more "buzz" or fanfare than articles or wiki pages. Thus, if there is a need to make a project's community aware of something, a blog post may be the best-suited medium.
- Writing can be time consuming, in particular for technical articles and blog posts. In addition, those media forms may need to undergo a review process. To get content out quickly, the wiki may be the best solution. However, readers may only find wiki pages if they are pointed to them explicitly.
- To solicit feedback from readers, articles and blog posts typically offer more comment functionality than the official product documentation or the wiki.

Offer Developers a Medium with a Low Entry Barrier for Fast Externalization of Content

Developers often prefer to externalize content in the form of wiki pages because wikis are easy to add to (J-L1, J-D1, J-D8). To encourage documentation, it is important that developers have such a medium at hand where they can externalize knowledge without having to worry about the accuracy of content or whether customers will understand the entire context. While it is useful to have other documentation artifacts, such as articles or blog posts, undergo a review cycle, developers need a platform for producing content with a lower entry barrier.

Involve the Community in a Project's Documentation

Unlike other forms of documentation, community portals allow for community involvement. While the number of documentation contributors outside the development team is still limited (J-M, J-D8), there have been successful instances: *"I got someone [...] who had been asking me a lot of questions and I said, well, you write an article. I had a draft ready, but that was still a skeleton and I said, here's a draft, go write your own style to it. It took a while, [...] but he ended up taking all the screenshots, taking my text and doing stuff and I just reviewed it."* (J-A) In particular, articles on topics such as best practices could be written by client developers and end users (J-M). Such contributions could even be encouraged with incentives as they provide considerable value to the community portal (J-M). Review processes would have to be in place to ensure quality. However, giving write access to community members is not realistic for all content, and in particular, not for the wiki. Other ways to increase community involvement could involve more options for marking up and commenting on content. In general, the entry barrier for community members to contribute to the documentation could be lowered (J-D1) by making the process more obvious and by offering easily-accessed web forms for contributions. Recent additions to jazz.net, such as commenting and voting features for library content, are steps in the right direction (J-L3), but more could be done to increase community involvement.

Provide Readers with an Option to Give Feedback

Feedback on documentation often comes through direct communication channels. Readers try to contact the content author via email or chat rather than ask questions on the forum. While this was not intended, it underlines the importance of the social aspect of documentation. Readers want to know who authored certain content and they want to be able to contact those individuals. However, with different systems for blog, technical articles, official product documentation, and the wiki, it is difficult to locate all contributions of a single individual.

This study found that readers take advantage of social media mechanisms to give feedback, such as rating of content or "liking" artifacts using Facebook Connect. These mechanisms have the potential to involve more readers and gather feedback on the quality and usefulness of content. In addition, implicit feedback, such as view counts, should be made use of.

6.1.6 Limitations

The first limitation of this study lies in the number of interviewees. However, I compared data from the interviews with artifact data from the community portal and with ethnographic field notes. Also, the interviewees had a range of different backgrounds, from managers to relatively new team members.

The choice of research methods – conducting semi-structured interviews, collecting ethnographic field notes, and web scraping of documentation artifacts – does not enable a precise characterization of what motivates developers to contribute to different kinds of documentation. While the interviews allowed for general insights into why developers choose different documentation formats, a detailed analysis of each document creation event is needed to gain a precise understanding of what motivates developers to create documentation.

While the documentation landscape for RTC was fairly stable at the time of this study, new projects, such as Rational Insight, had been added to jazz.net shortly before this study. This study focused on RTC as the mature part of jazz.net, but it is impossible to rule out that other projects might have influenced the findings.

6.1.7 Summary

In this case study of a software project community portal, I found that developers need traditional artifacts, such as the official product documentation, and social media artifacts, such as wiki pages or blog posts, to externalize different kinds of content. Each of these artifacts play a role in capturing knowledge that can help users, client developers, and team members understand a system. Using social media artifacts, developers often externalize content without being prompted by formal processes. They respond to questions, promote their part of the system or themselves, support development processes, or contribute through organized efforts.

Unlike traditional forms of documentation, social media artifacts have the potential to involve a community composed of client developers and end users in the process of externalizing developer content. In the next chapter, we leave the context of a closed source project, such as IBM's Jazz, and examine the role of social media artifacts in documentation on websites that are mostly concerned with open source software.

Chapter 7

Prevalence of Crowd Documentation

Social media has changed how software developers produce and consume documentation. While the Internet has made a vast body of knowledge accessible to software developers, it is the advance of social media that has introduced effective mechanisms for a large crowd of developers to curate content on the web. For example, in the case of Wikipedia, a large group of individuals come together to create and curate content on the web using social media technologies. Despite the lack of formal mechanisms to ensure the quality and comprehensiveness of content, Wikipedia has now become the *de-facto* standard for encyclopedias [29]. In other areas of the web, users can endorse articles through mechanisms, such as Facebook's "Like", they can give positive or negative ratings to questions and answers on Q&A websites, and they can annotate and comment on a wide spectrum of blog posts. Content on social media sites for software developers ranges from tutorials and experience reports to code snippets and samples.

Similar to how open source development disrupted the process of traditional software development [126], social media has the potential to redefine how developers learn, preserve, and communicate knowledge about software development. But unlike community documentation, where a person may freely contribute to the documentation of an open source project [42], with *crowd documentation*, the individual contribution through social media is not as important as the aggregate result. An individual in a crowd may make a one-time contribution, such as voting on an answer, tagging a question, providing an answer, or asking a question with minimal effort and little commitment. For developers who contribute to these new media, there is almost no barrier to entry, no community vetting, and there are no formal processes that would be otherwise associated with traditional or community-based forms of documentation.

7.1 Prevalence of Crowd Documentation in Google Search Results

To examine whether crowd documentation has the potential of replacing more traditional forms of documentation, we report on a study aimed at measuring the prevalence of documentation via social media. We selected one particular API – jQuery – and performed web searches for each of the API methods. We then examined the first 10 results for each API method and analyzed the different information sources available to developers. We found that 87.9% of the API methods are covered by software development blogs, and that these blog posts mainly consist of tutorials and reports on experiences using those API methods.

The remainder of this section is structured as follows. In Section 7.1.1, related work on API documentation is reviewed, before introducing our methodology in Section 7.1.2. Our findings are presented and discussed in Sections 7.1.3 and 7.1.4. The limitations of this study are described in Section 7.1.5, and this study is summarized in Section 7.1.6.

7.1.1 Related Work: API Documentation

Effectively documenting and using Application Programming Interfaces (APIs) is not an easy undertaking. Robillard [143] observed insufficient or inadequate examples and issues with the API's structural design as obstacles for developers trying to learn it. He also identified several other task, format and design-related obstacles.

These shortcomings have called the software engineering research community to action. But rather than improving the process of creating API or framework documentation, the solution to bad documentation has been instead to build better knowledge discovery tools [51]. Research tools, such as Hipikat [185] or Strathcona [93], provide developers with recommendations in the form of relevant artifacts or code examples. Jungloids [121] are based on the idea that programmers often know what type of object they need, but not how to access it. Jungloids support this process by automatically discovering how to convert from a set of initial types to a desired type. Jadeite [165] takes an alternative approach of displaying the most common way to construct a desired type by letting users add new "pretend" classes or methods that are displayed in the actual API documentation and can be annotated with the appropriate APIs to use.

Researchers have also considered improving API documentation by adding information on API usage. Holmes and Walker [94] used data on the popularity of API methods to recommend certain methods. Mica [164] augments standard search results to help programmers find API methods and examples of their usage by analyzing the content of web pages and classifying the results. Assieme [91] is a web search interface that supports programming search tasks by combining information from libraries, API documentation and pages with sample code and explanations to help programmers reduce the number of queries they have to run.

7.1.2 Research Methodology

This section presents our research questions as well as the methods we used for data collection and analysis.

Research Questions

Our research questions focus on the coverage of API methods on the web and on the characteristics of the corresponding resources. When our initial analysis revealed that blog posts played a prominent role in the search results, we focused on blog posts for our more detailed research questions.

1. Where are API methods documented on the Internet and how high is the API coverage of different kinds of resources?
2. How are API methods covered using blog posts?
3. Who writes blog posts about API methods?
4. How much interaction do blog posts about API methods elicit?
5. How frequently are source code snippets used in API-related blog posts?

Data Collection

To retrieve the resources that developers have at their disposal when doing a web search for API documentation, we performed web searches for all methods of the jQuery API, a popular cross-browser JavaScript library. At the time of our study, there were 173 API methods available. We performed the web searches using Google, ensuring that we were signed out of our personal accounts to avoid personalization of the search results. All queries were prefaced by "jQuery" (e.g., to retrieve the results for jQuery's `.add()` method, we searched for "jQuery add").

We then extracted the links to the first 10 search results for each method, yielding a total of 1,730 links. For each search result, we recorded the query that found it, the rank it had on the result page, the link, and the domain.

Data Analysis

We analyzed all 1,730 websites in our data using a mix of quantitative and qualitative methods. In a first step, we classified them using criteria such as the URL (e.g., for links starting with stackoverflow.com), or the content of the website. We collaboratively coded a subset of the links to check for consistency in our coding, and then divided the rest of the data set up for individual coding.

In a second step, we did additional qualitative coding of all 376 unique blog posts in our data set. We used the same coding process as in the first step, but focused on the content and style of the blog post. In addition to a label for the type of blog post, we recorded the number of code snippets used and the number of comments for each post.

7.1.3 Findings

This section first presents our findings on the prevalence of crowd documentation in Google searches, before focusing our investigation on blog posts in particular.

RQ1: Coverage by Different Kinds of Resources

The classification of the top 10 search results for all API methods revealed several different categories of web resources. The most frequent categories are given in Table 7.1 along with the extent to which they cover the different methods in the API

Table 7.1: API coverage by different web resources

search result type	coverage	mean rank
Official API	99.4%	1
Blog Post	87.9%	5
Stack Overflow	84.4%	6
Unofficial Documentation	63.6%	6
Official Forum	37.0%	3
Official Documentation	30.1%	3
Mailing List Entry	25.4%	7
Official Bug Tracker	21.4%	3
Forum	20.2%	8
Q&A	9.8%	9
Code Snippet Site	8.7%	9

and their mean page rank. The following paragraphs describe the different types of resources.

Official API. Part of the official API documentation, either documenting one of the API methods or listing several API methods belonging to the same category. For 99.4% of the API calls[1], the official API documentation was part of the first 10 search results and usually appeared on top.

Blog Post. A blog post, belonging to a blog with regular posts and a commenting feature. We found that there were blog posts among the top 10 search results for 87.9% of the API methods. Blog posts usually appeared right after the official documentation, and before Q&A websites, mailing list entries, and unofficial documentation. Details on the characteristics of the blog posts are given in the next section.

Stack Overflow. A Q&A thread on Stack Overflow appeared on the first page of the search results for 84.4% of the calls.

Unofficial Documentation. Method documentation found on websites other than the official website of the API came up for 63.6% of all API calls (often cloned from the official documentation, but potentially outdated).

Official Forum. For 37.0% of the API calls, threads in the developer forum hosted on the API website were part of the top 10 search results.

[1]All API calls except for `ajaxSuccess`.

Official Documentation. Official tutorials and other documentation material hosted on the API website only appeared for 30.1% of the API calls.

Mailing List Entry. Websites serving as entry point to the archive of a mailing list were part of the search results for 25.4% of the API calls.

Official Bug Tracker. For 21.4% of the API calls, a bug in the official bug tracking system came up in results.

Forum. A thread in a developer forum not hosted on the official API website appeared for 20.2% of the API calls.

Q&A. A Q&A thread on a website other than Stack Overflow appeared for 9.8% of the API calls.

Code Snippet Site. Code snippet sharing websites, such as CodeSnipr[2], had a coverage of 8.7%.

Other notable resources that appeared were the web-based hosting site GitHub and documentation in languages other than English.

RQ2: Different Kinds of Blog Posts

Table 7.2: Types of blog posts

post type	frequency
Tutorial	217
Experience	99
Code Snippet	35
New Functionality	27
Link / Referral	24
Opinion	12
Research	6
Announcement	6
Workaround	5
Repost	5
Design Idea	5

The qualitative coding of all blog posts in our data revealed the distinct categories shown in Table 7.2. The most common blog post types were tutorials and experience

[2]http://www.codesnipr.com/

reports by the blog authors. The following paragraphs describe the different post types.

Tutorial. The post contextualizes a problem or task and then proceeds to describe how to achieve a solution in a step by step manner.

Experience. The post documents development knowledge drawn from a recent experience, e.g.: *"I spent over a half hour looking for the best solution to this. Personally, I blame the jQuery documentation. When reading over the jQuery core description it states, starting in version 1.4, that jQuery returns an empty set but offers no method to detect it. Ultimately, I found that .length is the way to go but I wanted to expound on all three methods I discovered."*[3]

Code Snippet. The post provides a succinct piece of code easily copied and extractable, but has no additional explanation.

New Functionality. The post discusses new features in the latest release of a software.

Link / Referral. The post shares interesting news by linking to at least one other online resource.

Opinion. The post presents a technical opinion on an issue, design, or technology stack.

Research. The post describes an empirical investigation, such as performance between different functions, or compatibility across different browsers.

Announcement. The post announces a new release or updates to a release of a software.

Workaround. The post presents a solution to an encountered problem.

Repost. The post duplicates content from another source (often Stack Overflow) without adding additional content.

Design Idea. The post describes a novel technique or the merits of a particular approach to a problem.

[3]http://b-knox.com/181/detect-an-empty-set-in-jquery/

RQ3: Small Contributions from Many Authors

Many developers contributed to the documentation, even if they only authored a few posts. Over half the posts (214) came from authors contributing only 1 or 2 blog entries. In this sense, the documentation was in fact a product of the crowd, with 210 authors accounting for a total of 376 blog posts.

Other authors contributed more, with the top five publishing between 10 and 26 posts. Figure 7.1 shows the total distribution of posts by authors.

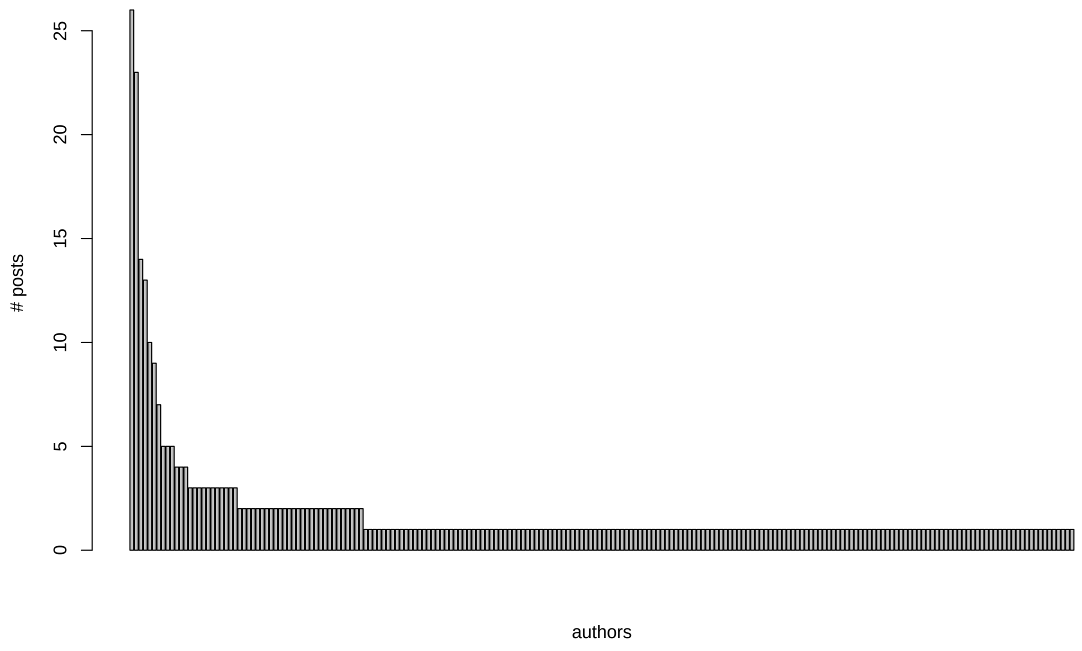


Figure 7.1: Number of posts contributed by authors

RQ4: Readers have Conversations with Authors

Some readers were actively engaged in conversation with the authors according to our data. 81% of the posts had at least one comment, with a median of 8 comments per post. Figure 7.2 shows a clearer picture of the distribution[4].

[4]We did not check for spam comments in the data.

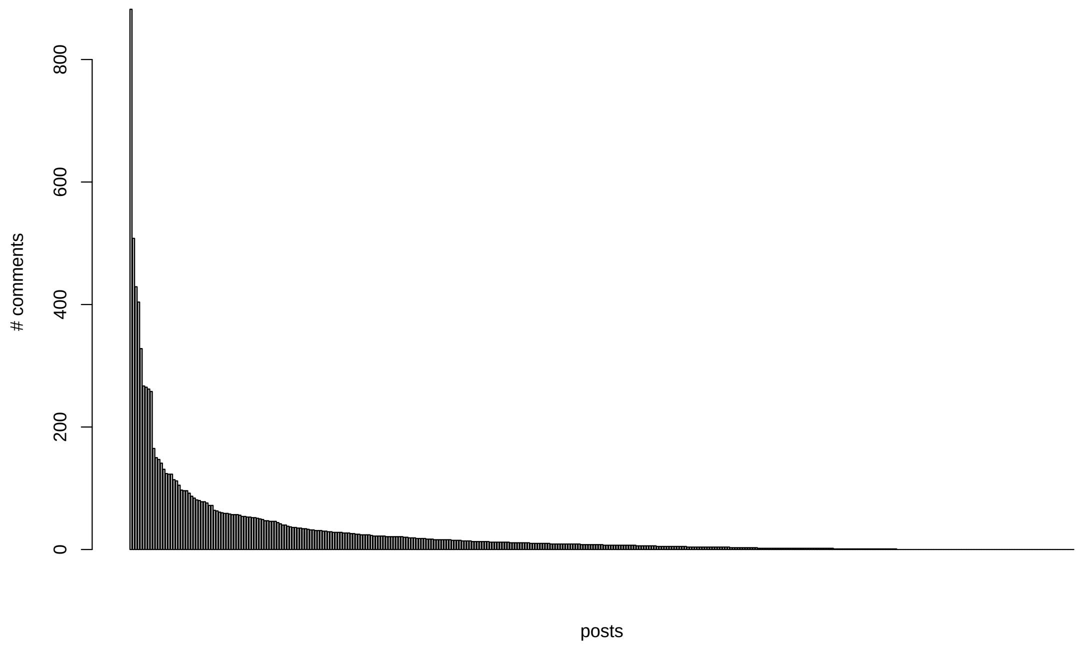


Figure 7.2: Number of comments per blog post

RQ5: Blogs Talk about Code

90% of posts had code snippets in the post, with a median of 3 code snippets per post. In Figure 7.3, the distribution is shown.

The presence of code snippets reinforces that posts are technical discussions featuring source code combined with insights, motivations, and design trade-offs. This is interesting to contrast with other web results, such as code snippet sharing sites. The 15 code snippets we observed from CodeSnipr had no explanatory text to explain the behaviour or purpose of the code.

7.1.4 Discussion

Based on our findings, this section discusses the potential motivations of the *crowd* to author documentation.

There were a few bloggers that may have deliberately sought advertising revenue but did offer legitimate content. These sites contained various advertisements that interrupted the flow of the blog posts. The content was not copied from any apparent source, such as the jQuery documentation, but the explanation was limited. One such

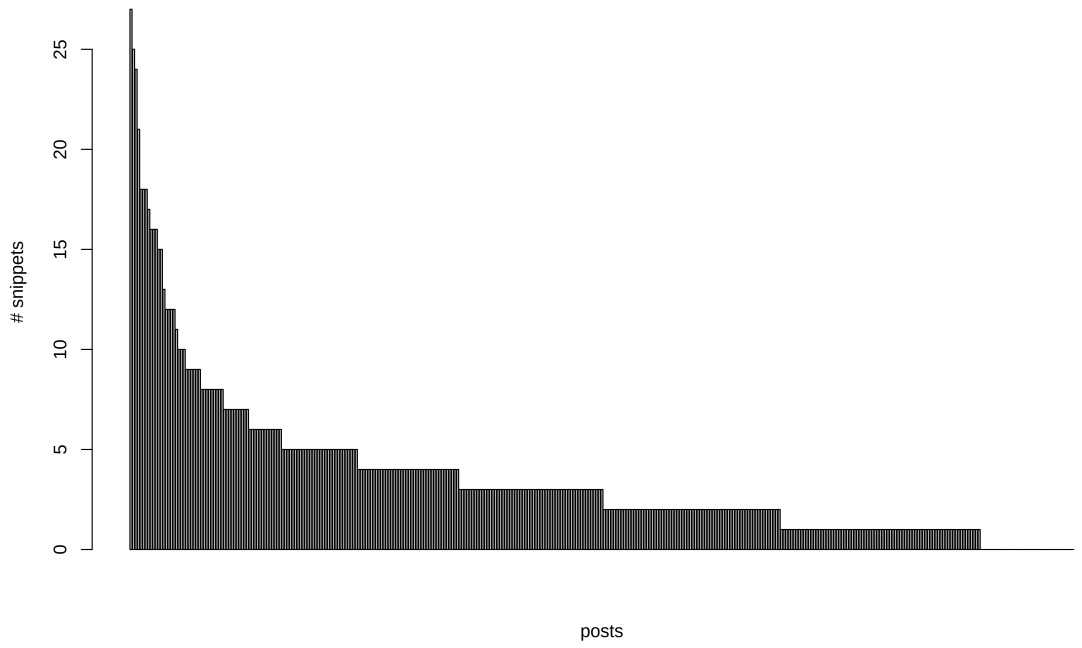


Figure 7.3: Number of code snippets per blog post

blogger had 9 posts comprising of several little tutorials, but did not go into much depth. His posts still frequently appeared in search results and received comments from readers.

In direct contrast with the previously described bloggers were powerhouse bloggers. These bloggers wrote extensively, featured a clean and professional website, and had a large following from readers and subscribers. One such blogger wrote many little tutorials. They were based on recent experiences, illustrated with real-world problems, and often were accompanied with video demonstrations that walked through and described the code. The blogger's 23 posts received 731 comments, and the author actively participated in the discussion with his readers. As this blogger runs an independent consulting firm, his motivations may be different from that of simple advertisement revenue – building a his professional reputation or personal branding may be much more valuable.

Finally, there were 158 authors who each contributed a single post. With such a diverse crowd, there are many different reasons that motivate the authoring of a blog post. For example, one author's contribution described an experience in resolving a conflict with other libraries, and how one of the jQuery API options did not resolve the problem as advertised. The author posted his workaround. We do not know what benefit the author gained from sharing this experience, but one reader shared his appreciation: *"I ran into a conflict of 2 libraries, did a few hours of researching and this was the only solution that worked for me. I don't quite understand it being new to jQuery, but it works! I'll drop a line in my code comments back to your site. Much appreciated."*

7.1.5 Limitations

In this study, we examined social media from the perspective of only one API. Although this provides a good starting point, the conclusions we draw may not hold true in other areas. As we studied web resources discussing web programming, we may have attracted developers interested in using web technologies. To address this limitation, we focus on one social media site – Stack Overflow – in the following sections, and we investigate its impact for several areas of software development.

Web search can be limited by a mismatch of semantic intention between the search terms and results. For example the search term, "jquery first", brings up blogs illustrating the function `first`, but also "my first jquery plugin", which may be interesting or irrelevant to the searcher.

Finally, we began our study with a limited set of search terms, in particular the names of API functions from jQuery. Developers may use a variety of search terms to target the questions or problems they have about programming. To address this, in the following, we focus on the Q&A website Stack Overflow and investigate the discussions on that site in more detail.

7.1.6 Summary

Our findings indicate that social media artifacts play an important role in software documentation, and that media, such as blog posts, can reach a high level of coverage featuring tutorials and personal experiences. The high coverage is a first indicator that social media is more than a niche in software documentation and that it can reshape the way software knowledge is communicated.

Chapter 8

Crowd Documentation on Stack Overflow

The conceptualization of crowd documentation as described in the previous chapter raises the following question: what is documentation? According to Parnas, we *"must recognize that there are many types of documents, each useful in different situations."* [134]. Crowd documentation can take different forms and use different media channels, from blogs and wikis to Q&A websites. With the rise of documentation via social media channels, software documentation is no longer limited to manuals and help files produced by technical writers. Any blog post and any answer on a Q&A website can now be considered documentation.

One of the most successful social media resources for software developers is Stack Overflow, a site that facilitates the exchange of knowledge between software developers through a Q&A portal. Since its foundation in 2008, more than 2.3 million questions have been asked on Stack Overflow, and nearly 5 million answers have been provided, contributing to a large repository of software development knowledge. Many of the questions and answers contain code snippets that can easily be integrated into one's source code.

Despite such widespread use of Q&A websites, the role of Q&A websites in the software documentation landscape is not well understood. It is unclear what kinds of questions get answered well and which ones remain unanswered, as well as how to phrase questions effectively. We do not know how helpful the questions and answers are to a broader community nor what the implications are from the duality of a Q&A website as a question and documentation portal.

A question asked on Stack Overflow has a median answer time of 11 minutes [120]. This virtually real-time access to a community of millions of other programmers willing and eager to help is an enormously popular resource, evidenced by the more than 12 million visitors and 135 million page views which Stack Overflow receives each month[1].

To understand the role of crowd documentation on Stack Overflow, we have conducted research on the nature and impact of the documentation created by the crowd. In a qualitative analysis of content on Stack Overflow, we found that social media is particularly effective at providing code reviews and answers to conceptual questions (Section 8.1). These findings indicate that social media is more than a niche in documentation, that it can provide high levels of coverage (Section 8.2), and that it gives readers a chance to engage with authors.

8.1 Nature of Crowd Documentation

The success of social media has introduced new ways of exchanging knowledge via the Internet. Q&A websites, such as Yahoo! Answers[2], Answers.com[3], or the recently created Facebook Questions[4], are founded on the success of social media and built around an "architecture of participation" [129] where user data is aggregated as a side-effect of using Web 2.0 applications. Questions and answers on Q&A websites represent archives with millions of entries that are of value to the community [20].

In this section, we report results that identify the role of Q&A websites in software development using qualitative and quantitative research methods. Our findings, obtained through the analysis of archival data from Stack Overflow and qualitative coding, indicate that Q&A websites are particularly effective at code reviews, explaining conceptual issues, and answering newcomer questions. The most common use of Stack Overflow is for how-to questions, and its dominant programming languages are C#, Java, PHP and JavaScript. Understanding the processes that lead to the creation of knowledge on Q&A websites will enable us to make recommendations on how individuals and companies, as well as tools for programmers, can leverage the knowledge and use Q&A websites effectively.

[1]http://www.quantcast.com/stackoverflow.com
[2]http://answers.yahoo.com/
[3]http://www.answers.com/
[4]http://www.facebook.com/questions/

The remainder of this section is structured as follows. Related work on the role of questions and answers with regard to software development is summarized in Section 8.1.1, and Stack Overflow is described in more detail in Section 8.1.2. Our methodology is introduced in Section 8.1.3, and our findings on the different kinds of questions on Stack Overflow are presented in Section 8.1.4. These findings are discussed in Section 8.1.5, and the limitations of this study are outlined in Section 8.1.6 before summarizing this study in Section 8.1.7.

8.1.1 Related Work: Question and Answer Websites and Software Engineering

Answer Garden by Ackerman and Malone [2] was an early exploration of an "organically" growing Q&A system that used questions and answers about the X Window System for its initial database. More recently, the advance of social media has made Q&A websites widely accessible and popular. In particular the use of Yahoo! Answers has been studied by several researchers. Gyöngyi *et al.* [79] identified three fundamental ways in which Yahoo! Answers is used: for focused questions, to trigger discussions, and for random thoughts and questions. Adamic *et al.* [3] found that users who focused on certain areas of expertise often received the best ratings. In order to find high quality content, Agichtein *et al.* [4] introduced a framework that is able to separate high quality items from the rest. In a related project, Shah and Pomerantz [157] found that contextual information, such as a user's past interactions, can be used to predict content quality.

While researchers have not yet studied what kind of questions software developers ask using Q&A websites, there has been work on questions asked by software developers in general. Letovsky [110] identified five main question types: why, how, what, whether and discrepancy. Fritz and Murphy [65] provide a list of questions that focus on issues that occur within a project. Sillito *et al.* [158] provide a similar list focusing on questions during evolution tasks. LaToza and Myers [107] found that the most difficult questions from a developer's perspective dealt with intent and rationale. In their study on information needs in software development, Ko *et al.* [101] found that the most frequently sought information included awareness about artifacts and coworkers.

In contrast to the settings of these studies, Q&A websites provide a platform for questions aimed at a general audience that is not part of the same project. Q&A

websites not only contain questions, but can also contain answers to anticipated questions as well as opinions through comments and ratings.

8.1.2 The Stack Overflow Portal

Stack Overflow is centered around several design decisions[5]: Voting is used as a mechanism to highlight useful answers. Users can up-vote answers they like, and down-vote answers they dislike. In addition, the user asking a question can accept one answer as the official answer. Tags are used to organize questions. Users must attach at least one tag and can attach up to five tags when asking a question. The editing of questions and answers allows users to improve their quality and to turn Q&A exchanges into wiki-like structures. Badges are given to users rewarding them for their contributions once they reach certain thresholds. Pre-Search helps avoid duplicate questions by showing similar entries as soon as a user has finished typing the title of a question. Stack Overflow was designed to be used such that Google is the main entry point to the site, and web pages are optimized towards search engines and performance. To ensure critical mass, several programmers were explicitly asked to contribute in the early stages of Stack Overflow.

8.1.3 Research Methodology

This section describes our methodology by outlining our research questions as well as our data collection and analysis methods.

Research Questions

We pose two research questions about the role of Q&A websites in software development.

1. What kinds of questions are asked on Q&A websites for programmers?
2. Which questions are answered and which ones remain unanswered?

Data Collection

Our data collection follows a mixed-methods approach, collecting quantitative and qualitative data. We created a script to extract questions along with all answers, tags

[5]http://www.youtube.com/watch?v=NWHfY_lvKIQ

and owners using the Stack Overflow API. We extracted two weeks worth of data, and the amount of data that we extracted is provided in Table 8.1.

Table 8.1: Data extracted from Stack Overflow

data item	amount
Questions	38,419
Owners	31,729
Answers	68,467
Tag instances	111,408

Data Analysis

To answer our research questions, we analyzed quantitative properties of questions, answers and tags, and we applied qualitative codes to a sample of tags and questions. Qualitative coding was done by two researchers individually at first, and codes were then refined and grouped in collaborative sessions.

8.1.4 Findings

In this section, our findings to the two research questions are reported.

RQ1: Different Kinds of Questions

To analyze the different kinds of questions asked on Stack Overflow, we did qualitative coding of questions and tags. The tags were mainly used to learn about the topics covered by Stack Overflow, while the question coding gave insight into the nature of the questions.

Each question had between one and five tags that are set by the person asking it. Most questions (72.30%) had between 2 and 4 tags. 10,272 different keywords were used to tag questions, and there were 111,408 instances of a tag being applied to a question. Table 8.2 shows the most-used tags.

We applied qualitative coding to the 200 most frequently used tag keywords in our data. These keywords covered 60,193 of the tag instances (54.03%). We identified five categories of tags that are shown in Table 8.3. We show the number of instances of each category, the number of different keywords per category, and highlight the most-used tags per category as examples. We did not find any related tags for the keyword `homework`, and thus left it uncategorized.

Table 8.2: Most-used tag keywords on Stack Overflow

tag keyword	instances
`c#`	3,765
`java`	2,909
`php`	2,599
`javascript`	2,310
`jquery`	2,084

Table 8.3: Coding of tags on Stack Overflow

code	keyword	instances	examples
Programming Language	63	28,218	`c#`, `java`
Framework	48	11,532	`jquery`, `ruby-on-rails`
Environment	45	14,127	`android`, `iphone`
Domain	29	4,125	`regex`, `database`
Non-functional	14	2,071	`multithreading`
Homework	1	120	`homework`

Users self-code their questions through tags to index their questions and to allow for navigation to them. Tags reveal the topics covered on Stack Overflow, but only allow limited insights into the nature of the questions asked. To further understand the characteristics of questions on Stack Overflow, we coded a random sample of 385 questions from our data set (1%). We analyzed the titles and body texts of these questions and found the following categories, ordered by their frequency:

How-to. Questions that ask for instructions (e.g., *"How to crop image by 160 degrees from center in asp.net"*).

Discrepancy. An unexpected behaviour that the person asking the question wants explained (e.g., *"iphone - Coremotion acceleration always zero"*).

Environment. Questions about the environment either during development or after deployment (e.g., *"How to use windows emacs as a svn client"*).

Error. Questions that include a specific error message (e.g., *"C# Obscure error: file '' could not be refactored"*).

Decision help. Asking for an opinion (e.g., *"Should a business object know about its corresponding contract object?"*).

Conceptual. Questions that are abstract and do not have a concrete use (e.g., *"Concept of xml sitemaps"*).

Review. Questions that are either implicitly or explicitly asking for a code review (e.g., *"Simple file download via HTTP - is this sufficient?"*).

Non-functional. Questions about non-functional requirements, such as performance or memory usage (e.g., *"Mac - Max Texture Size for compatibility?"*).

Novice. Often explicitly states that the person asking the question is a novice (e.g., *"Oracle PL/SQL performance tuning crash course"*).

Noise. Questions not related to programming (e.g., *"Apple Developer Program"*).

Most questions in our random sample fit into one of these categories, but for some of the questions (9.61%), we assigned two categories. The most frequent type of question (39.22%) was how-to, followed by questions about discrepancies and environment. The first two columns of Table 8.4 show the detailed results of our coding.

Table 8.4: Coding of questions on Stack Overflow

		answered		no
code	sum	accepted	not accepted	answer
How-to	151	67 (44%)	63 (42%)	21 (14%)
Discrepancy	50	27 (54%)	11 (22%)	12 (24%)
Environment	40	13 (33%)	17 (43%)	10 (25%)
Error	36	19 (53%)	14 (39%)	3 (8%)
Decision help	22	9 (41%)	10 (45%)	3 (14%)
Conceptual	18	10 (56%)	7 (39%)	1 (6%)
How-to/Novice	16	10 (63%)	3 (19%)	3 (19%)
Review	13	12 (92%)	1 (8%)	0 (0%)
Non-functional	10	6 (60%)	1 (10%)	3 (30%)
Novice	5	2 (40%)	3 (60%)	0 (0%)
Other	24	10 (42%)	11 (46%)	3 (13%)
Sum	385	185 (48%)	141 (37%)	59 (15%)

RQ2: Which Questions Are Answered and Which Are Not

Figure 8.1 shows the distribution of answers per question. The number of answers per question is shown on the x-axis, and the number of questions with that number of answers is shown on the y-axis using a log scale. 5,450 (14.19%) questions were not answered. The remaining questions had at least one and up to 23 answers. Only 3,243 out of 68,467 answers (4.74%) were provided by the same person that had asked the question.

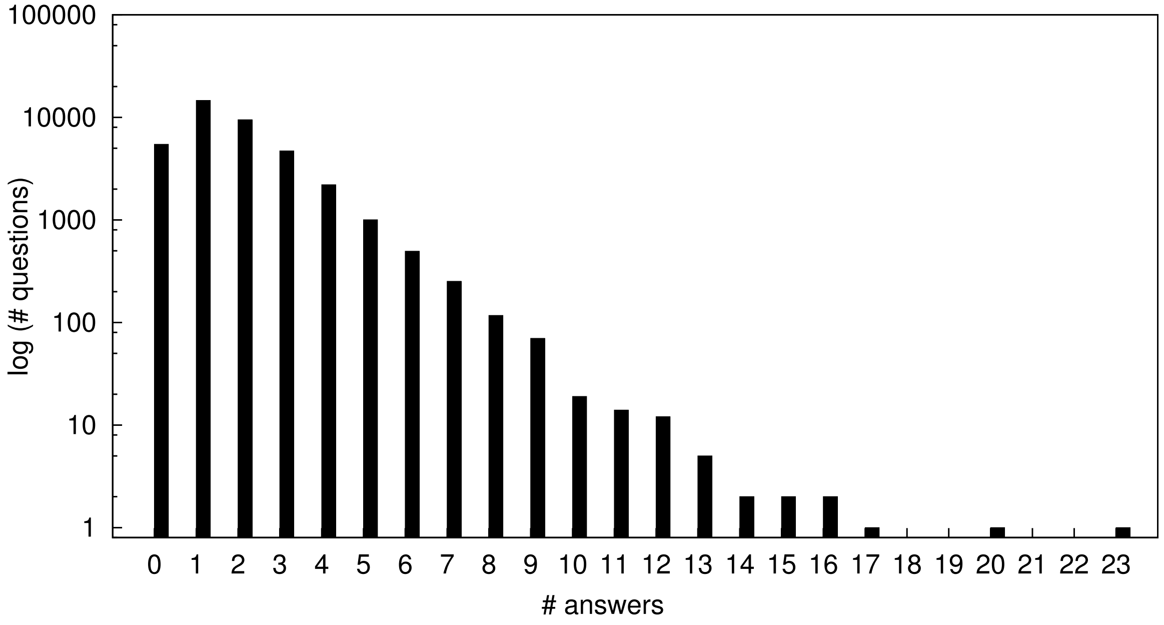


Figure 8.1: Answers per question

On Stack Overflow, the user who is asking a question can mark a maximum of one answer per question as *accepted*. We use this feature to examine the implications of different question characteristics on the success of a question. We define successful and unsuccessful questions as follows:

> A successful question has an accepted answer, and an unsuccessful question has no answer.

Following these definitions, we further analyzed the 185 successful questions and 59 unsuccessful questions from our random sample of 385 questions. Table 8.4 shows the number of questions per category for all questions in our random sample, for all successful questions, for all questions with answers but no accepted answer, and for all questions without an answer.

It is interesting to note that the community answered review, conceptual and how-to/novice questions more frequently than other kinds of questions.

8.1.5 Discussion

A possible reason for the high answer ratio of review questions is the fact that review questions are often very concrete. They contain code snippets, and often no external sources are necessary to understand the code and make a recommendation about its quality. Also, code review questions can have more than one “correct” answer, and often any input is better than no input. The knowledge required to answer conceptual questions is usually broad, and it is available in documentation or books and only needs to be presented effectively. Novices are easy to sympathize with and their questions are usually easy to answer.

The type of question is not the only factor for eliciting good answers. Other factors include: the technology in question, the identity of the user, the time and day in which the question was asked, whether the question included a code snippet, or the length of the question.

8.1.6 Limitations

The first limitation lies in the small amount of data we analyzed in our random sample. However, by triangulating our findings through qualitative coding of tags and questions, we were able to mitigate some of these concerns. Our definitions of successful and unsuccessful questions are narrow, but they do offer a first approximation. Our conclusions are limited to Stack Overflow and its current use. However, Stack Overflow has managed to attract a large group of users and is one of the first Q&A websites specifically for software developers.

8.1.7 Summary

Understanding the interactions on Q&A websites, such as Stack Overflow, will shed light on the information needs of programmers outside closed project contexts and will enable recommendations on how individuals, companies and tools can leverage knowledge on Q&A websites.

Our findings indicate that Stack Overflow is particularly effective for code reviews, conceptual questions and for novices. The most common questions include how-to questions and questions about unexpected behaviours. The next section describes a study we conducted to investigate the impact of the crowd in the software documentation landscape, focusing on Stack Overflow.

8.2 Impact of Crowd Documentation

To illustrate the potential impact of crowd documentation: while the official documentation for Java provides one code example[6] to demonstrate synchronization with `invokeLater()`, but without explanatory text and sparse comments, 286 related questions can be found on the Q&A website Stack Overflow. Not only are the contributions on Stack Overflow more numerous, but the crowd also provides explanatory text, succinct code snippets, multiple perspectives debating the merits of different solutions, votes on popular answers and good questions, and tags for improving searches.

To explore the feasibility of crowd documentation for software development, we examined the coverage of the discussions about APIs on Stack Overflow. We collected evidence from over 7 million question and answers on Stack Overflow, accumulated over a 4 year span, and studied in detail: 6,323 questions and 10,638 answers related to the Google Web Toolkit (GWT) API[7], 119,894 questions and 178,084 answers related to the Android API[8], and 181,560 questions and 445,934 answers related to the Java API[9]. Usage data for 2,432 classes of the Java API and for 1,038 classes of the Android API through Google Code Search[10] was also collected.

We found that crowd documentation can generate many examples and explanations of API elements. Although the crowd can achieve a high coverage of API elements, the speed of achieving that coverage was linear over time. Furthermore, sometimes the crowd may need to be steered towards certain topics. For example, the crowd was mute on certain topics, such as accessibility (e.g., accommodate disabled users) or digital rights management (DRM) capabilities in Android.

The remainder of this section is structured as follows. Our methodology is described in Section 8.2.1, and our findings are presented in Section 8.2.2. Section 8.2.3 discusses our findings, Section 8.2.4 discusses the limitations of this study, and the study is summarized in Section 8.2.5.

8.2.1 Research Methodology

This section outlines our research questions as well as our data collection and analysis methods.

[6]http://java.sun.com/developer/technicalArticles/Threads/swing/Editor.java
[7]http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/index.html
[8]http://developer.android.com/reference/packages.html
[9]http://docs.oracle.com/javase/6/docs/api/
[10]http://www.google.com/codesearch

Research Questions

To determine the value of crowd documentation, it is crucial to understand whether we can rely on the crowd to provide questions and answers for an entire API on Stack Overflow in a reasonable amount of time. To answer this question, several aspects must be considered: the extent to which the crowd discusses different API elements, which areas of an API are covered (and which ones are not), and the speed at which this coverage is achieved.

1. Are APIs widely covered?
2. What is discussed and what is not discussed?
3. How fast is the crowd at producing documentation that covers an entire API?

Data Collection

To retrieve data from Stack Overflow, we downloaded the Creative Commons Data Dump that the Stack Overflow team makes available on their blog[11] and imported it into a relational database. For each question and each answer on Stack Overflow, we obtained information such as the title and body, the tags assigned to the question, the creation time stamp as well as the time stamp of the last edit, and the number of times somebody had viewed the contribution.

We examined the crowd documentation of three popular APIs: the Android API, the GWT API, and the API of the Java programming language. These APIs were selected because they have different characteristics, and yet are easy to compare. While all are written in Java, the Java API is large and well-established, whereas the Android API is young and only applies to the domain of mobile development. Finally, while the GWT and Android APIs are comparable in size, they have different levels of activity on Stack Overflow. For all APIs, we downloaded a list of all packages and classes: GWT had 845 classes, Android 1,038 classes, and Java 2,432 classes.

In addition, usage data for the APIs using Google Code Search was collected. For each class in our data set, we queried Google Code Search for import statements containing that particular class (e.g., to get usage data for the Android class TextView, we queried Google Code Search for "import android.widget.TextView"). Since most modern code editors support automatic import organization, this offers

[11]http://blog.stackoverflow.com/category/cc-wiki-dump/

a good approximation of actual usage data. Generic imports, such as "import android.widget.*", are rarely used[12]. Unfortunately, we could not collect usage data for GWT because of the recent closure of the Google Code Search API.

Data Analysis

To determine whether a Q&A thread on Stack Overflow talks about a particular class, we used case-sensitive, word boundary matches. Additionally, several semantic rules were applied to ensure correct matches. In the case of name collisions, we used the fully qualified name. For example, for the Java `Date` class, we used fully qualified names to distinguish `java.util.Date` and `java.sql.Date`. There were a few cases where word boundaries did not correctly distinguish classes. For example, in Android, using naïve word boundaries, `AbsListView.SelectionBoundsAdjuster` did also match `AbsListView`. To avoid this problem, we used look ahead matching in certain cases. Finally, to avoid collisions with common English words, we excluded links for single-word API elements, such as the `Intent` class in Android, and only considered them when they were part of a code snippet.

8.2.2 Findings

In the following, we present the findings on whether we can rely on the crowd to discuss an entire API on Stack Overflow.

RQ1: Extent of Coverage by the Crowd

To answer our first research question as to whether APIs are widely covered, we examined the crowd documentation's coverage of API elements. *Coverage* is the percentage of classes that have at least `n` threads discussing the class, where `n` is called the saturation level. To understand how coverage varied across different APIs, we examined the coverage of the three selected APIs on Stack Overflow: GWT, Android, and Java. Figure 8.2 shows the results.

For 87% of all classes of the Android API as well as for 77% of all classes of the Java API, we found at least one thread on Stack Overflow. Although the number of Java classes is more than twice the number of Android classes, both reached a comparable level of coverage. In contrast, even though GWT is only slightly smaller

[12]For example, Google Code Search finds 52,790 import statements with the fully qualified name of TextView compared to 2,023 search results for the generic import of the android.widget package.

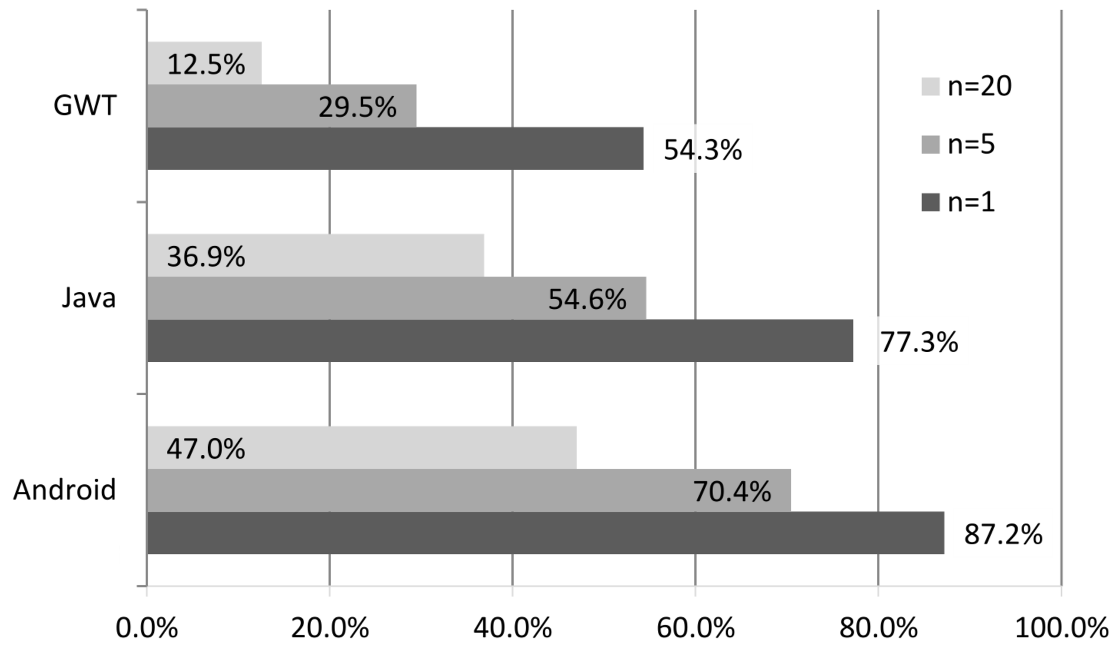


Figure 8.2: Coverage of API elements

than Android, it has a lower level of coverage at each saturation level. Low activity on Stack Overflow can impact the comprehensiveness of crowd documentation – only 3,277 threads discussed GWT API classes in comparison to 70,123 threads that discussed Android API classes. Finally, coverage with higher levels of saturation (i.e., at least 20 threads per class) is lower, but still comes in at 47% for Android, 37% for Java, and 12% for GWT.

RQ2: Areas Covered by the Crowd

Our second research question asked what is discussed on Stack Overflow, and what is not. We examined this question in two steps: In a first step, we analyzed the correlation between how often certain classes are used and how often they are mentioned on Stack Overflow to understand whether low coverage for particular elements can be explained by low usage in general. In a second step, we conducted a manual inspection of the most and least covered packages of each API to see what kind of classes are discussed a lot, and which ones are not discussed at all.

For Android and Java, we found a strong correlation between usage data (from Google Code Search) and coverage data (from Stack Overflow): A Spearman's rank correlation coefficient of 0.797 for Android, and a Spearman's rank correlation coefficient of 0.772 for Java. Classes that are used by many developers are likely to have a high discussion volume on Stack Overflow, and classes that are only used infrequently in practice are not likely to be well-documented by the crowd either. We were not able to obtain usage data for GWT because of the recent closure of Google Code Search.

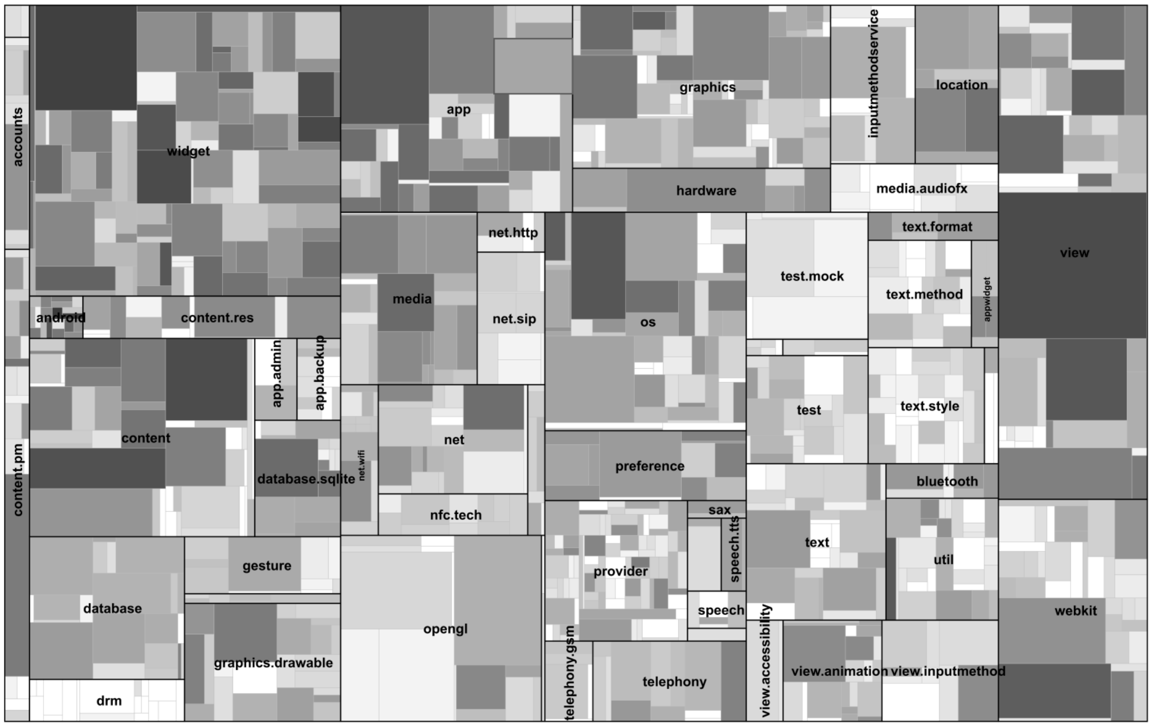


Figure 8.3: Crowd documentation of Android classes

To examine in more detail which areas are thoroughly discussed on Stack Overflow and which areas are largely ignored by the crowd, we developed a visualization that shows the number of threads for each class of an API. Figure 8.3 shows the result for the Android API. Packages and their classes are shown as a treemap, and the number of methods determines the size for each class. A logarithmic colour scale is used to indicate the number of threads: white classes have no threads whereas dark classes have many threads. Popular packages, such as android.widget and android.view, are well-covered, whereas areas such as the digital rights management (android.drm) and accessibility (android.accessibilityservice) are largely ignored by the crowd. An interactive version of the visualization is available online for Android[13] and Java[14]. For Java, popular packages, such as java.util and java.swing, are covered well, whereas areas such as java.security are largely ignored.

RQ3: Speed of the Crowd

To determine how fast the crowd is at covering an entire API, we examined the speed and coverage over time of crowd documentation. To measure speed, we looked at how many API elements were discussed at every date between July 31, 2008 and December 1, 2011 for Android, Java, and GWT. To measure coverage, we calculated coverage of classes with various saturation levels (n = 1, 5, 20, 50, 100 threads) over time.

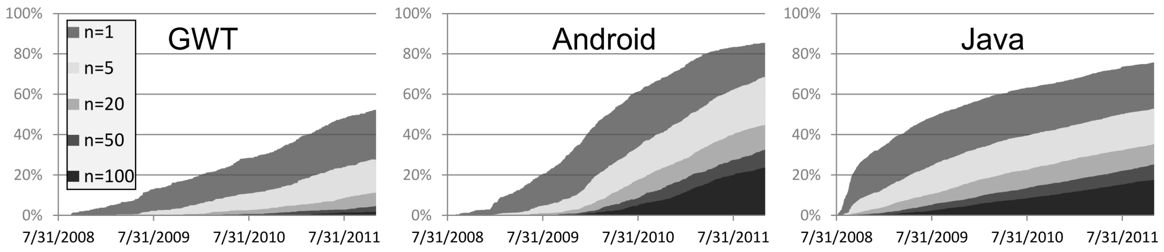


Figure 8.4: Coverage of API elements over time

The results for the three APIs can be seen in Figure 8.4. For all three APIs, the rate at which new classes are covered by the crowd with various saturation levels followed a linear pattern, with the exception of Java in the first year. The crowd was very quick to discuss new Java classes in the first year of Stack Overflow (about half of all classes in the Java API were mentioned on Stack Overflow by the end of July 2009), but the speed of the crowd decreased after that.

API designers cannot completely rely on the crowd to provide questions and answers for an entire API. While there is at least one Q&A thread on Stack Overflow for about 80% of the classes of the Java and Android APIs, some areas are ignored by the crowd. Also, the crowd takes time to discuss all classes of an API at a linear rate, focusing on API elements that are frequently used.

[13]http://latest.crowd-documentation.appspot.com?api=android/
[14]http://latest-java.crowd-documentation.appspot.com/?api=java

8.2.3 Discussion

This section discusses our findings and identifies implications of our research.

Steering the Crowd

We found the crowd to be too slow to ever replace official API documentation. For example, within the first year of Stack Overflow's operation, the coverage of Android API classes only reached 30%. After time, the situation greatly improved, but still left pockets of uncovered areas. However, there are several possible ways to help steer the crowd, for example, by injecting incentives into the crowd. Because participation can be driven by reward [120], API designers may be able to use these incentive channels to directly recruit the crowd to help them out. They can reward bounties (reputation points) for providing questions and answers to specific API packages or tasks. Badges (achievements that can be earned by Stack Overflow users) can automatically be rewarded to those that first ask or answer a question about a certain API element. Reputation can be a powerful incentive: Highly reputable Stack Overflow users have been contacted by software company recruiters based on their reputation, or received offers for consulting business.

Finally, API stakeholders trying to establish an API may find it worthwhile to detect and invest expertise when an insufficient number of expert contributions are being made.

Impact of Crowd Documentation

According to our data, the 307,774 threads from Android, Java, and GWT were viewed a total of 200 million times[15]. Combined with the data on the prevalence of crowd documentation in search results (see Section 7.1), the potential impact of crowd documentation on the software documentation landscape as a whole is significant. As we can see from the discussions of the online software developer community, crowd documentation through blogs, wikis, and Q&A websites has proven to be useful. Unlike traditional documentation, it lacks the authoritativeness that comes with an "official" nature, but it compensates for this weakness by providing a vast number

[15]While there is no official data available on the implementation of the view count on Stack Overflow, the implementation is *"very very strict"* according to co-founder Jeff Attwood, see http://meta.stackoverflow.com/questions/58369/did-anyone-notice-that-some-sites-seem-to-be-scraping-republishing-sos-posts/58374.

of resources and curation through a large crowd of contributors. As we have shown with our data on the prevalence of blogs and Q&A threads in search results as well as with the view counts of threads on Stack Overflow, software developers increasingly refer to documentation generated by the crowd.

Crowd Documentation Analytics

For better understanding of crowd documentation, we developed an interactive treemap visualization that can be freely configured to show the number of threads, usage, or number of methods for the classes of an API using size and colour mappings. In addition, the visualization can be filtered by user name to highlight the contributions of a particular user. This allows researchers and API designers to understand their community and even allows users to visualize their contributions amidst the crowd. For example, potential areas of difficulty, as identified by a disproportionate ratio of discussion in questions and usage in practice or by large patches of undiscussed sections of APIs, could be detected and explored.

8.2.4 Limitations

Linking API classes to threads on Stack Overflow is not a simple undertaking. Our analysis described here excludes a class name that was not matching the exact case (e.g., `jsonparser`) or with spaces placed in between words (e.g., `Zoom Buttons Controller`). In addition, to avoid collisions with common English words, we adapted our methodology for single-word API elements. This approach may have resulted in excluding some mentions of single-word elements outside of code snippets on Stack Overflow.

We focused on API elements that were classes and our results may not generalize to other elements, such as methods, or other aspects of APIs, such as installation or deployment.

We only examined our research questions from the perspective of three APIs. Although they provide different perspectives (a large established API, a new and specialized API, and a less active API), we do not know the extent to which our results extend to other APIs that are more difficult, different in size, or that fill a particular niche. We also did not have usage data for GWT, due to the recent shutdown of the Google Code search service.

8.2.5 Summary

We have shown several sources of evidence that crowd documentation exists as a viable process that can emerge from various social media sites, such as blogs and Stack Overflow. Documentation can emerge in the form of blog posts, questions and answers that feature many code examples and discussions about using API classes and methods, and the authors that contribute these items take distinct roles in curating and maintaining the quality of these documents.

Crowd documentation is an effort that is shared by many. Even if many only contribute a few items, the net result can achieve a high coverage of API functionality with a large impact reaching huge audiences. The process works with similar principles as open source development but is driven by unique and complex factors.

This concludes the findings on the role of social media artifacts in documentation. The next chapter describes a model of social media artifacts in collaborative software development that synthesizes the findings on task management and documentation.

Part IV

Implications

Chapter 9

A Model of Social Media Artifacts in Collaborative Software Development

Based on the empirical studies presented in Parts II and III, I developed a model of social media artifacts in collaborative software development. This model aims to answer the research questions on the role of social media artifacts, and how they interplay with traditional artifacts and mechanisms in software development. Six dimensions were identified along which social media artifacts differ from traditional artifacts in software development. This model aims to assist developers and managers in their decisions of adopting social media mechanisms, and to provide them with guidance on what characteristics distinguish social media artifacts from traditional artifacts.

The model presented is based on the model of artifacts in a community portal that was discussed in Section 6.1. This model is extended in a series of steps: Section 9.1 presents a model of social media artifacts in documentation which extends the original model by including Q&A threads, based on the findings presented in Chapter 7. Then, a model of social media artifacts in task management is presented in Section 9.2, based on the findings from Part II. In Section 9.3, the models are combined into the model of social media artifacts in collaborative software development. As the models are based on the empirical work presented in this thesis, they may not generalize beyond the teams under study, in particular not to development teams outside of IBM.

9.1 Social Media Artifacts in Documentation

Table 9.1 summarizes the model of social media artifacts in documentation. For each of the eight dimensions that emerged from the grounded theory study described in Section 6.1, the model shows the nature of each artifact – the official documentation, technical articles, blog posts, wiki pages, and Q&A threads – with respect to the dimension. Apart from the official documentation, all artifacts are considered as social media artifacts. The model is based on the empirical study with IBM's Jazz team presented in Section 6.1 as well as the research on crowd documentation presented in Chapters 7 and 8.

The next paragraphs again explain each dimension in more detail.

Content. One of the findings from the empirical study on the documentation practices of IBM's Jazz team described in Section 6.1 is that the official product documentation focuses on features rather than scenarios, whereas articles, blog posts, and wiki pages all contain scenarios to some extent. In addition, articles contain descriptions of how to achieve a certain task, and they typically provide more depth than the official product documentation. Blog posts contain case studies and personal views and are often used for marketing, whereas wiki pages contain plans, scenarios, instructions, and references to support articulation work (see Section 6.1). The content of Q&A threads was discussed in detail in Section 8.1: the most common types were questions about how to solve a certain task, followed by questions about discrepancies and the development or deployment environment.

Where traditional documentation (i.e., the documentation available through a help menu) typically focuses on *features*, the documentation available through social media artifacts contains *scenarios*.

Audience. The audience for the official product documentation are the customers[1] of that product, mainly because the documentation is shipped as part of the product. The same holds true for technical articles which are meant to communicate scenarios to customers. Blog posts are aimed at the wider project community, composed of customers, client developers, and the software development community in general. Wiki pages are used to communicate content to individuals inside of the development

[1]Customers for the projects under study, in particular IBM's Jazz, can be end users as well as client developers.

Table 9.1: Social media artifacts in documentation

artifact	content	audience	trigger	collab-oration	review	feedback	fanfare	time sensitivity
Official Doc	Features	Customers	Product releases	Explicit	Formal	None	Combined with release	Sensitive, not updated
Articles	Scenarios	Customers	Questions from users	Some	Semi-formal	Comments	Feeds	Can be updated
Blog Posts	Case studies	Community	Questions from users	Some	Semi-formal	Comments	Feeds	Can be updated
Wiki Pages	Scenarios	Developers, Customers	Articulation work	Some	Not formal	Edits	None	Can be updated
Q&A threads	How-to etc.	Community	Questions from users	Crowd	Not formal	Comments	Feeds	Can be updated
Social Media	Scenarios	Diverse	Questions from users, articulation work	Implicit	Semi-formal or less	Comments	Feeds	Can be updated
Tradi-tional	Features	Customers	Product releases	Explicit	Formal	None	Combined with release	Sensitive, not updated

team as well as to individuals outside of the development team (see Section 6.1). The audience of Q&A websites, such as Stack Overflow, is the entire community of software developers on the web.

Customers are the primary audience of traditional documentation, whereas the audience of social media artifacts is *diverse* and can include developers, customers, and the entire software development community.

Trigger. A new version of the official product documentation is triggered by a new product release, as the documentation is part of the product. Technical articles and blog posts are triggered by questions from users, by feature promotion, or by organized documentation efforts in a development team. A new wiki page is usually written as part of articulation work (see Section 6.1). According to Gerson and Star [72], *"articulation consists of all tasks needed to coordinate a particular task, including scheduling sub-tasks, recovering from errors, and assembling resources"*. A new Q&A thread is created whenever a user posts a new question.

Traditional documentation is produced because of *product releases*, whereas documentation through social media artifacts is created to *answer questions from users* and to support *articulation work*.

Collaboration. Collaboration around the production of the official product documentation is explicit, as the official product documentation is produced by a separate team of technical writers. Technical articles, blog posts, and wiki pages are mostly created in solo efforts, although some evidence was found for collaboration around these artifacts (see Section 6.1). The Q&A threads on sites such as Stack Overflow are produced by a crowd of developers asking questions, writing answers, adding comments, and voting on content.

The collaboration mechanisms around social media artifacts are *implicit*. While there is some collaboration, none of it is formally defined. On the other hand, collaboration around traditional documentation is *explicit*.

Review. Content in the official product documentation is reviewed rigorously, and the processes around its review are formalized. While there is still a review process in place for technical articles and blog posts, these artifacts are less formal and more personal. Content on wikis is rarely reviewed and informal (see Section 6.1). Content in Q&A threads does not undergo quality control, and is not formal.

Documentation in social media artifacts is either *semi-formal* or *not formal*, whereas traditional documentation is *formal*.

Feedback. Customers typically have no means to give feedback on the official product documentation. On the other hand, feedback mechanisms in the form of comments are available for technical articles, blog posts, and Q&A threads. The only way to give feedback on wiki pages is through edits, because there is no distinct comment section (see Section 6.1).

Feedback options typically depend on the tools that development teams use for documentation. For the teams in the studies presented here, the traditional documentation had *no feedback* option, whereas most of the social media artifacts allowed for feedback through *comments*.

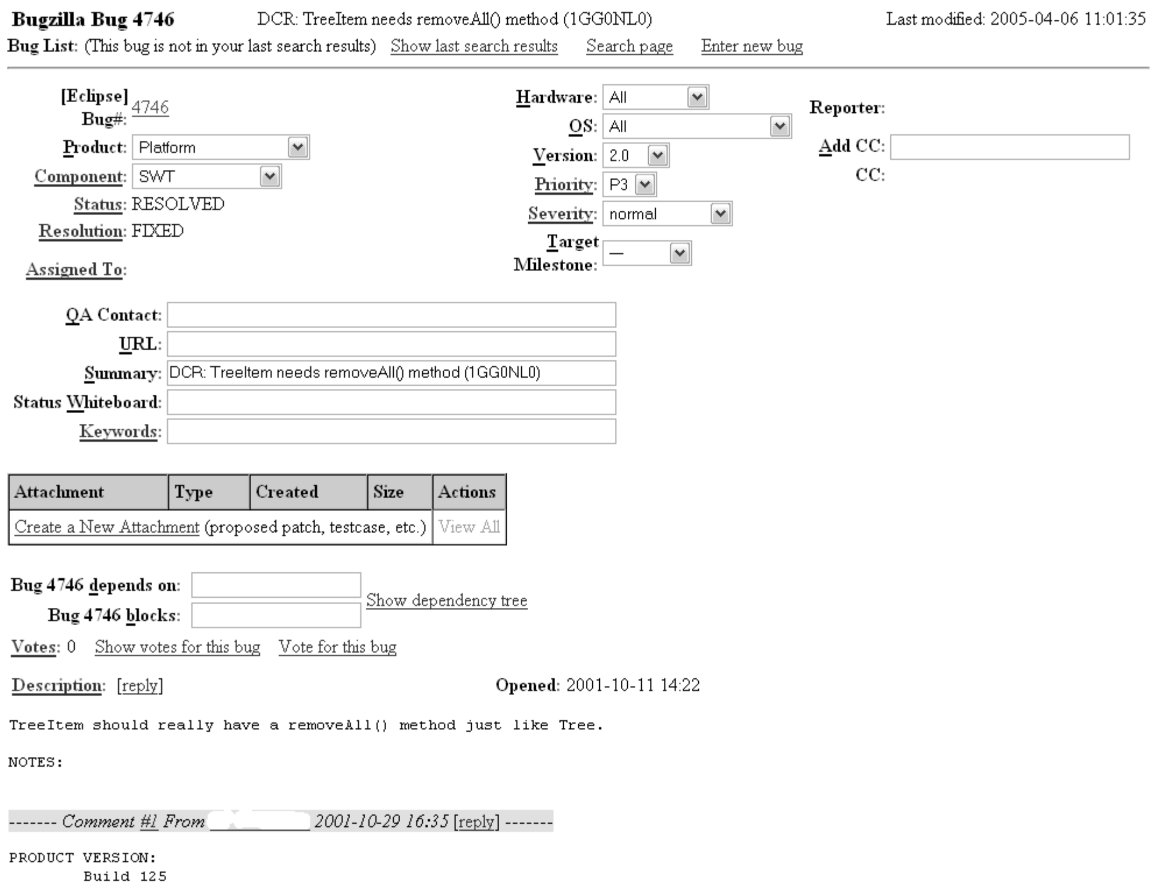
Fanfare. In the projects under study, in particular in IBM's Jazz, the amount of fanfare around a new version of the official product documentation is the same as the fanfare around new product releases. In other words, customers are usually aware when there is a new version of the official product documentation. Technical articles, blog posts, and Q&A threads are released with less fanfare, however, there are some feeds available that developers can subscribe to in order to be informed of new artifacts (e.g., through Twitter or RSS). New wiki pages do not have any fanfare associated with them (see Section 6.1).

Social media artifacts in software documentation can typically use a variety of *feeds* to keep subscribers up-to-date. On the other hand, traditional documentation can take advantage of the official *product releases* to make customers aware of new versions.

Time Sensitivity. The official product documentation is very time-sensitive. Due to its focus on features rather than scenarios, it becomes outdated as soon as a small part of the functionality is changed. There are also no update mechanisms in place. For all other artifacts, there is a way to update them after their original publication. Still, technical articles and blog posts are rarely updated in practice (see Section 6.1). Technical articles typically contain information that is related to a particular release, and they are written in a way that they do not become outdated quickly. Producing new technical articles and technical blog posts is time-intensive. Wiki pages can become outdated very quickly, but they are easily changed. Threads on Q&A websites can be updated in various ways: the original question or answer can be edited, old content can be given negative votes, and more up-to-date content can be given positive votes.

The distinguishing characteristic with respect to time sensitivity is whether there is an update mechanism in place: social media artifacts *can be updated*, while the traditional documentation is *not updated*.

9.2 Social Media Artifacts in Task Management



Bugzilla Bug 4746 DCR: TreeItem needs removeAll() method (1GG0NL0) Last modified: 2005-04-06 11:01:35

Bug List: (This bug is not in your last search results) Show last search results Search page Enter new bug

[Eclipse] Bug#:	4746	**Hardware:**	All	**Reporter:**	
Product:	Platform	**OS:**	All	**Add CC:**	
Component:	SWT	**Version:**	2.0	**CC:**	
Status:	RESOLVED	**Priority:**	P3		
Resolution:	FIXED	**Severity:**	normal		
Assigned To:		**Target Milestone:**	—		

QA Contact:
URL:
Summary: DCR: TreeItem needs removeAll() method (1GG0NL0)
Status Whiteboard:
Keywords:

Attachment	Type	Created	Size	Actions
Create a New Attachment (proposed patch, testcase, etc.)				View All

Bug 4746 depends on:
Bug 4746 blocks: Show dependency tree

Votes: 0 Show votes for this bug Vote for this bug

Description: [reply] **Opened**: 2001-10-11 14:22

```
TreeItem should really have a removeAll() method just like Tree.

NOTES:
```

------- *Comment #1 From* 2001-10-29 16:35 [reply] -------

```
PRODUCT VERSION:
        Build 125
```

Figure 9.1: Traditional task management with Bugzilla [11]

Table 9.2 summarizes the model of social media artifacts in task management in comparison to traditional task categories, such as priority and severity. The main difference between social media artifacts and traditional task categories is that the former can be freely configured by all users of the task management system whereas

Table 9.2: Social media artifacts in task management

artifact	content	audience	trigger	collaboration	review	feedback	fanfare	time sensitivity
Traditional	Features	Developers	New release or component	Explicit	Formal	Comments	Feeds	Not sensitive
Annotations	Concerns	Individual, Team, Community	Articulation work	Explicit, Implicit	Not formal	None	Feeds	Release-specific
Dashboards	Concerns	Managers, Developers	Articulation work	Implicit	Not formal	None	None	Release-specific
Feeds	Task updates	Developers	Articulation work	Implicit	Not formal	None	Feeds	Not sensitive
Social Media	Concerns	Diverse	Articulation work	Implicit	Not formal	None	Feeds	Release-specific
Traditional	Features	Developers	New release or component	Explicit	Formal	Comments	Feeds	Not sensitive

the latter can typically only be configured by an administrator. For example, every developer can create a new tag keyword to annotate a task, but only administrators are able to add a new priority or severity level. For traditional task management approaches, administrators – a role that is typically fulfilled by the development manager – are "gatekeepers" who can unilaterally decide what features are available for the management of tasks. With social media approaches, these features emerge out of decisions made by all developers. Most modern task management systems, such as Bugzilla[2], contain traditional task management features as well as social media artifacts.

Traditional task management for software developers includes pre-defined categories, free-form text, attachments, and dependencies for each task (see Figure 9.1 for an example of a task in Bugzilla). The pre-defined categories offer a variety of data about each task. Some values, such as the task ID, creation date and creator, are assigned when the task is created. Other values, such as the product, component, operating system, version, priority and severity, are selected by the developer who creates the task, but can be changed over the lifetime of the task. Other categories routinely change over time, such as the person to whom the task is assigned, the current status of the task, and if resolved, its resolution state [11].

For each of the dimensions, the model in Table 9.2 shows the role of categories in traditional task management, the role of annotations (i.e., tags and labels), and the role of dashboards and feeds. The findings are based on the work presented in Part II. The labels and key-value pairs described in the context of Google Code are discussed together with tags described in the context of IBM's Jazz because they are used in similar ways, as was outlined in Chapter 5. The implications for both artifacts are grouped under "annotations". The next paragraphs explain each dimension in detail.

Content. Most of the categories used in traditional task management for software development are focused on features or defects. The task ID, creation date, and creator provide details about the creation of the task, and the product, component, and operating system categories put the task into context with other features. The version number puts it into context of the release history, and the priority and severity categories are used to classify tasks according to the concerns of priority and severity, and these concerns influence the scheduling of tasks. Articulation work is supported by categories such as the person to whom the task is assigned, the current status

[2]http://www.bugzilla.org/

of the task, and if resolved, its resolution state. Annotations on the other hand are primarily used to communicate a wide range of concerns, from cross-cutting concerns to planning concerns (see Sections 3.1 and 5.1). Dashboards are then used to surface these concerns (see Section 4.1), and feeds contain any sort of task update.

The focus of traditional task management categories, such as component and operating system, is on *features* or defects. Social media artifacts, on the other hand, focus on *concerns*, and provide various ways of attaching these concerns to tasks and surfacing them.

Audience. The audience of traditional task management categories and feeds are the developers on a project. For annotations, the situation is less clear: tagging is done for the team and a wider community (see Section 3.1, as well as related work on task annotations [161]). Similarly, dashboards are aimed at developers as well as at managers, and they are accessible to the customers of a project (see Section 4.1).

Social media artifacts in task management have a *diverse* audience, whereas traditional approaches focus on *developers*.

Trigger. Changes to categories in traditional task management systems are typically triggered by a new release or by the development of a new component. New annotations are created to support articulation work, in particular categorization and organization (see Section 3.1). New dashboards are created for a number of reasons: to gain an overview of the project status, for peripheral awareness, to identify bottlenecks or to compare teams, and for navigation and tracking of particular tasks. All of these activities can be characterized as articulation work. The same applies to feeds: they are mainly used to track work at a small scale, as a personal inbox, and for planning purposes (see Section 4.1).

Changes to traditional task management categories are triggered by *new releases or components*, while social media artifacts in task management are triggered by *articulation work*.

Collaboration. Collaboration processes in traditional task management are explicit. The various categories specify who created a task, who currently owns it, and who is in charge of quality insurance (QA, see Figure 9.1). Collaboration processes around annotations can be explicit or implicit, and focus on tolerance rather than explicit rules (see Section 3.1). Similarly, collaboration processes around dashboards are mostly implicit. While the most-used dashboards are the ones on team and project levels, there are no explicit processes governing the collaboration around them. Feeds are neither created through explicit collaboration (see Section 4.1).

Traditional task management categories focus on making collaboration processes *explicit*, whereas social media artifacts in task management foster *implicit* collaboration.

Review. From the developers perspective, traditional task management categories have a *"very administrative-side feeling"* where it is impossible *"to just* ad hoc *make things."* (J-D3, see Section 3.1). Annotations on the other hand are not formal, and they are rarely edited or reviewed. Similarly, neither dashboards nor feeds are formal (see Section 4.1).

Social media artifacts in task management are *not formal*, whereas traditional task management is *formal*.

Feedback. The availability of feedback mechanisms in task management depends on the particular implementation of the task management system. For the artifacts in these studies, there was a *comment* feature available for traditional task categories in IBM's Jazz and on Google Code, but not for any of the social media artifacts (see Part II).

Fanfare. Similarly to feedback, the amount of fanfare with which new artifacts are announced depends on the particular task management system. In these studies, there were *feeds* for new tasks and changes to task categories as well as annotations, but not for dashboards (see Part II).

Time Sensitivity. Traditional task categories are not time sensitive. In fact, they can be seen as being ephemeral because tasks are moved into an archive as soon as they are resolved. Unless a developer specifically queries the archive, resolved tasks and their categories do not appear in search results or in default views. Annotations and dashboards on the other hand are release-specific. In particular, the planning-related annotations often contain the name of a particular release in their name (e.g., `beta2candidate`, see Sections 3.1 and 5.1). Dashboards typically contain pages for particular releases that need to be cleaned up manually after each release. Feeds are ephemeral by nature, and only display the latest events. They do not get re-configured between releases (see Section 4.1).

While traditional task management categories are *not time-sensitive*, social media artifacts often carry *release-specific* information which makes them time-sensitive.

9.3 Social Media Artifacts in Collaborative Software Development

Table 9.3 summarizes the task management and documentation models and shows how social media artifacts differ from traditional artifacts in collaborative software development along the eight different dimensions.

Traditionally, software development tool and process support has focused on features. Social media provides opportunities to communicate concerns and scenarios to a diverse audience composed of developers, managers, customers, and a wider community. The creation of traditional artifacts, such as task categories or help manuals, is triggered whenever a developer creates a new component or whenever a product is released. These triggers are explicit steps in the development process. On the other hand, social media artifacts are created because of user questions or to support articulation work. Collaboration using traditional development processes and tools is explicit, whereas collaboration around social media artifacts is implicit. Similarly, social media artifacts are less formalized than traditional artifacts. In terms of time sensitivity, social media artifacts are release-specific but they usually offer update mechanisms and can thus be kept up-to-date. Traditional artifacts become outdated quickly, but they often do not surface once they are out-of-date.

Both social media artifacts and traditional artifacts can make use of feeds to create fanfare, and for both kinds of artifacts, there are commenting features available. Feedback and fanfare depend on the particular tools used, and it appears not to be a characteristic of social media as much as it is a characteristic of a particular tool solution.

Based on this model, I define the role of social media artifacts in collaborative software development as follows:

> The role of social media artifacts in collaborative software development is the (1) timely dissemination of (2) scenarios and concerns (3) to a diverse

Table 9.3: Social media artifacts in collaborative software development

artifact	content	audience	trigger	collab-oration	review	feedback	fanfare	time sensitivity
Social Media	Scenarios, concerns	Diverse	Questions from users, articulation work	Implicit	Semi-formal or less	Comments?	Feeds?	Can be updated, release-specific
Tradi-tional	Features	Clearly defined	Process trig-gers, articu-lation work	Explicit	Formal	Comments?	Feeds?	Outdated quickly

audience through a process of (4) implicit and (5) informal collaboration, (6) triggered by questions from users or articulation work.

Chapter 10

Conclusions and Future Work

Social media mechanisms, such as wikis, blogs, tags and feeds, have transformed the way we create and curate content online. For example, despite the lack of formal mechanisms to ensure the authoritativeness of the subject matter, the wiki-based website Wikipedia has now become the *de-facto* standard for encyclopedias. Flickr, an online photo management and sharing application, successfully employs the social media mechanism of tagging to have their photo collection organized by a large crowd of users. This approach of crowd-sourcing – outsourcing a task to the general public – has proved successful for many projects, ranging from the collaborative mapping project OpenStreetMap[1] to the marketplace for human intelligence tasks, Amazon Mechanical Turk[2]. All of these are successful examples of large groups of contributors synthesizing their efforts to create and curate content online without formal rules and processes, using a variety of social media mechanisms.

In this thesis, I have examined how social media mechanisms can support one collaborative process in particular: software development. For a long time, tool and process support for software developers focused on providing highly formalized processes and systematic tool support for the individual software engineer. However, highly formalized tooling often stands in contrast to constantly changing requirements in a changing software development landscape, where customers expect software to be developed *"faster, better, and cheaper"* [188] and to run on a variety of devices, from desktop computers to smartphones and tablet computers.

Traditionally, software development tools and processes lacked the informal and social channels that would allow them flexibility to adapt to changing requirements

[1]http://www.openstreetmap.org/

[2]https://www.mturk.com

and a changing landscape [45, 85]. In recent years, social media mechanisms have made their way into tools and processes for collaborative software development, and they are starting to reform the way developers organize and document their work. This thesis has presented how social media artifacts, such as tags, feeds, and dashboards, can bridge lightweight and heavyweight task management in software development, and have illustrated how blogs, developer wikis and Q&A websites are changing the way software is documented.

Based on the results of several empirical studies, I found that the role of social media artifacts in collaborative software development lies in the the timely dissemination of scenarios and concerns to a diverse audience through a process of implicit and informal collaboration, triggered by questions from users or articulation work. Tool and process support inspired by social media can enable software development teams to adapt to changing requirements, and to address the challenges involved with having to coordinate and communicate with hundreds of other developers on any given project. Social media provides new opportunities to involve a project's community through crowd-sourcing of processes such as documentation.

The remainder of this chapter is structured as follows. Section 10.1 summarizes the contributions from this thesis, and the limitations of the research methods are discussed in Section 10.2. Section 10.3 outlines future work and Section 10.4 concludes this thesis.

10.1 Contributions

The overarching goal of this research is to advance collaborative software development by studying current practices and by innovating tools inspired by social media. To that end, the research questions have focused on understanding the role of social media artifacts in collaborative software development, on developing tools that leverage the knowledge from social media artifacts, and on analyzing the interplay of social media artifacts with other development artifacts. I make four contributions:

Several large scale empirical studies of software development teams.

Empirical software engineering aims to better understand the practice of software engineering by treating software engineering as an empirical science [137]. Several empirical studies were conducted with various professional software development teams to understand the role that social media artifacts play in these teams. Through a variety of methods for data collection, ranging from ethnographic-style observations and semi-structured interviews to mining software repositories (MSR), I gained insight into how and why developers use mechanisms that have been inspired by social media. Grounded theory [37] was employed to analyze the data that was collected.

A model of social media artifacts in collaborative software development. Based on the empirical studies, a model of social media artifacts in collaborative software development was developed (see Chapter 9). This model distinguishes traditional software development artifacts from social media artifacts along six dimensions: the type of content typically represented in the artifact, the audience for which the artifact is intended, the motivation that triggers the creation of a new artifact, the extent of collaboration during the creation of a new artifact, the extent to which new artifacts are reviewed before publication, and the time sensitivity of information in the artifact. I conclude that the role of social media artifacts in collaborative software development lies in the timely dissemination of scenarios and concerns to a diverse audience through a process of implicit and informal collaboration, triggered by questions from users or articulation work.

Tool and process recommendations. Empirical research grounded in studies with professional software developers has the power to affect change in the processes and tools that software development teams and organizations use. Several of the tool and process recommendations have been implemented after the studies. While I have no conclusive evidence that there is a causal relationship between the recommendations and their implementation, the number of changes suggests that the empirical work presented in this work has influenced the tools and processes of the teams under study. One of the major recommendations from the work on task management was to add tags for artifacts other than development tasks (see Part II). In IBM's Jazz, tags are now available for builds[3] and items in the jazz.net library are also organized using tags. Another recommendation was the improvement of search functionality and management of tags. Both of these recommendations have been addressed in a recent release; queries on tags can now be run more easily and bulk edits have been

[3]https://jazz.net/help-dev/clm/topic/com.ibm.team.build.doc/topics/ttagbuild.html

introduced[4]. The findings related to documentation (see Part III) led to actionable advice by articulating the benefits and possible shortcomings of the various communication channels. Two of the recommendations – involving the community[5] and clearly marking internal content as draft[6] – have since been implemented by the team under study.

New tools for developers that leverage social media artifacts. Based on the research on task management, two tools were developed that leverage the information stored in social media artifacts. The first tool, ConcernLines, supports the cognitive process of understanding how information about the release history, non-functional requirements, and project milestones relates to functional requirements on software components by visualizing co-occurring tags over time (see Section 3.2). The second tool, WorkItemExplorer, enables developers and managers to investigate trends and correlations in their task data by making exploration flexible and interactive, and by utilizing multiple coordinated views. Our evaluation of WorkItemExplorer with professional developers showed that the tool is able to answer questions developers ask, while enabling them to gain new insights through the free exploration of data (see Section 4.2).

10.2 Reflections on Research Methods

The limitations for each of the studies were discussed as part of reporting the results in Parts II and III. To discuss my choice of research methods in general, I refer to the four criteria for validity that Lincoln and Guba [113] define for qualitative work:

Credibility. To establish credibility of qualitative research, Lincoln and Guba recommend member checking as well as prolonged engagement in the field to determine whether the findings “make sense”.

Transferability. Lincoln and Guba encourage researcher to provide a detailed portrait of the setting in which the research is conducted in order to give readers enough information to judge the applicability of the findings to other settings.

[4]https://jazz.net/blog/index.php/2010/12/03/whats-new-in-rational-team-concert-3-0-part-v-work-item-enhancements/

[5]https://jazz.net/library/submit

[6]https://jazz.net/wiki/bin/view (account required)

Dependability. Dependability can be achieved by providing documentation of the data, methods, and decisions of a research project.

Confirmability. The confirmability of qualitative research can be enhanced through triangulation of data as well as through documentation of the research methodology.

To increase credibility of the findings, the results of the work related to IBM's Jazz have been presented to the developers under study on several occasions. Some of the developers have also read pre-prints of the papers that correspond to the chapters in this thesis. The findings related to Stack Overflow have been presented to Joel Spolsky, one of the creators of Stack Overflow, in an email conversation. In addition, for all studies, I have provided a detailed portrait of the research setting, and I have discussed the data collection and analysis methods as well as their justification in detail. This should enable the transferability of this work as well as ensure its dependability.

The work on tagging has been replicated by a group of researchers from Italy [31]. In addition to replicating the study with IBM's Jazz, the group considered tagging in the task management system Trac[7], and confirmed the findings presented in Section 3.1.

The conclusions are limited to the particular implementations of social media functionality in IBM's Jazz, Google Code, and Stack Overflow. However, IBM's Jazz is one of the first development environments to support social media artifacts, such as tags, dashboards, and feeds, and it is one of the first platforms that opened up its development process to the community without being open source, and therefore provides a unique case between open source and traditional closed source projects. Similarly, Stack Overflow provides a unique case as it has managed to attract a large group of users and is one of the first Q&A websites specifically for software developers. For task management and documentation, I studied teams using IBM's Jazz as well as developers using other tools (such as Google Code and Stack Overflow), to further solidify the findings presented here. Also, throughout this work, focus was placed on social media artifacts, such as tags, blogs and feeds, that are widely used in software development, and that are not unique to the systems under study.

I cannot claim generality beyond the scope of the particular projects studied. In particular teams developing software for other domains, such as security, might

[7]http://trac.edgewall.org/

use social media artifacts differently, or not at all. Privacy and security concerns might prohibit developers from using informal mechanisms as part of their tools and processes.

As more projects adopt tools inspired by social media, such as IBM's Jazz, Google Code or Stack Overflow, or as other development environments adopt social media mechanisms, additional studies should be conducted to gain further insight into the role of social media artifacts in collaborative software development.

10.3 Future Work

Social media is changing how software developers manage their tasks and how they produce and consume documentation. The current adoption of social media artifacts in processes and environments is just scratching the surface of what can be done by incorporating social media approaches and technologies into collaborative software development. Social media opens up new opportunities, but it comes with challenges for developers and researchers [13]. While I have gained some understanding of the particular processes that social media can support, this research opens up many avenues for future work which are outlined below.

The impact and risks of social media. Social media implies that processes and tools are socially open, and that content can be used in several different contexts [129]. This stands in contrast to many traditional software practices, such as closed source and formal team hierarchies. In future work, I propose to study the impact of using social media tooling in software development. How can projects balance an "architecture of participation" and individual productivity? What can management learn from collaboration artifacts, and how can social media data be aggregated in meaningful ways? Which parts of development processes are not captured in artifacts? How can privacy concerns be mitigated, and what risks are related to software developers using social media? Through empirical studies, researchers can develop theories that explain the impact of social media, and that can inform the development of social media tools that aim to improve software engineering practices.

Collaboration beyond development teams. The ultimate goal of commercial software development is the creation of business value. Software development is not an isolated undertaking in a company, but part of a bigger enterprise that involves many stakeholders, such as support, marketing, legal, and accounting. While these stakeholders do not need to understand the intricacies of source code or builds, it is important that they comprehend the value created by software development in order to optimize their investment strategies. They should also recognize the risks involved, such as cost overruns or failure to deliver on time. To provide stakeholders with awareness of value and risks, I propose to develop tools and processes that allow for insights at a higher level than current awareness tools. Stakeholders in management roles are currently not considered by many of the tools and platforms for software developers. In the work on awareness mechanisms inspired by social media, I investigated the role of dashboards and feeds in collaborative software development and found that development managers and developers had different uses for these mechanisms (see Section 4.1). In future work, I propose to build on this work by studying and innovating tools and techniques that support collaboration between all stakeholders.

Social media beyond software development. Just as social media has largely emerged outside of software development and then found its way into the work practices of developers[8], other disciplines can learn from the adaptations that software developers have made to social media tools and processes. The results of this work, for example, can be transferred into areas outside of software development. The work on the interplay of formal and informal approaches to task management can inform the management of any set of artifacts, such as in content management systems. The results on documentation in software development are applicable to communication of content through the web in general. With the multitude of communication channels available (e.g., blogs, wikis, Twitter, Facebook), we often struggle to find the right channel to publish through. Innovations in social media often occur when large groups of users co-opt technology for their own needs and purposes, thus creating *de-facto* standards. In future work, I propose to continue using the rich repositories of social media tools and technologies to study the role of social media in various contexts, and to innovate tools and processes that are grounded in empirical research.

Social media artifacts in other processes and tools. For this thesis, the work

[8]With the exception of wikis, see http://c2.com/cgi/wiki?WikiHistory.

has been scoped to the processes of task management and documentation. While I have gained a good understanding of the role of social media in these processes, there are many other processes in software development where the informal nature of social media may be beneficial. An example is requirements engineering, the process of discovering, documenting and maintaining a set of requirements for a given software project. In a typical software development project, requirements constantly change [34], and modern development methods, such as Agile, advocate an iterative and incremental development, where requirements and solutions evolve over time [90]. Using a mixed-methods approach similar to the one used in this thesis, in future work, I propose to study the role of social media in requirements engineering. Furthermore, new web-based tools for software developers inspired by social media are constantly being innovated. In future research, we can set out to understand the role of developer portals such as GitHub, CodeSnipr, and Masterbranch. How does "social coding in GitHub" [40] enable transparency and collaboration in software development? How do software developers manage their reputation online on Masterbranch? What meta-data do code snippets on CodeSnipr need to be considered useful? Future work will determine the role of these portals, and contribute to our understanding of how these new mechanisms can interplay with the tools that developers have used before the rise of social media.

10.4 Concluding Remarks

Social media has transformed the way communicate, work and play online. Without formal mechanisms to ensure the quality and comprehensiveness of content, websites such as Wikipedia and Facebook have become *de-facto* standards that we rely on for knowledge acquisition and social interactions. In addition, services such as Twitter are not only documenting but possibly reinforcing world events, such as the Arab Spring. All of these examples show how large groups of individuals collaborate effectively without formalized rules or processes, using social media as a vehicle.

In this work, I have studied the role that social media can play in addressing challenges in collaborative software development. Many software development teams struggle to succeed in an environment that is shaped by constantly changing requirements and a changing software development landscape. Developers often lack informal communication channels as well as tools and processes that bridge the technical and social aspects of development work.

Through several large scale studies with various professional development teams, this work has shown that a wide range of social media mechanisms have been adopted and adapted by software developers, and that they help in bridging the gap between social and technical aspects of development. Social media allows for the timely dissemination of scenarios and concerns to a diverse audience through implicit and informal collaboration, and it stands in contrast to many traditional software practices, such as closed source and formal team hierarchies.

As social media continues to become a ubiquitous part of our lives, it is not just another tool in the toolbox of software developers. Rather, entire development processes and tools are becoming social. This paradigm shift will democratize development processes by lowering the boundaries between software organizations as well as between developers and their customers, it has the potential to provide new ways for the various stakeholders in software development to collaborate, and it can enhance and improve development processes by addressing the many challenges facing collaborative software development.

Bibliography

[1] A. Acar and Y. Muraki. Twitter for crisis communication: lessons learned from japan's tsunami disaster. *International Journal on Web Based Communities*, 7(3):392–402, 2011.

[2] M. S. Ackerman and T. W. Malone. Answer garden: a tool for growing organizational memory. In *Proceedings of the Conference on Office Information Systems*, COCS '90, pages 31–39, New York, NY, USA, 1990. ACM.

[3] L. A. Adamic, J. Zhang, E. Bakshy, and M. S. Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 665–674, New York, NY, USA, 2008. ACM.

[4] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne. Finding high-quality content in social media. In *Proceedings of the International Conference on Web Search and Data Mining*, WSDM '08, pages 183–194, New York, NY, USA, 2008. ACM.

[5] A. Aguiar, G. David, and M. Padilha. Xsdoc: an extensible wiki-based infrastructure for framework documentation. In *Proceedings of the 8th Jornadas Ingeniería del Software y Bases de Datos*, JISBD '03, pages 11–24, 2003.

[6] N. Ahmadi, M. Jazayeri, F. Lelli, and S. Nesic. A survey of social software engineering. In *Proceedings of the International Conference on Automated Software Engineering - Workshops*, ASE '08, pages 1–12, Washington, DC, USA, 2008. IEEE Computer Society.

[7] K. R. Al-asmari and L. Yu. Experiences in distributed software development with wiki. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and*

Compilers, SERP '06, pages 389–393, Las Vegas, Nevada, USA, 2006. CSREA Press.

[8] S. W. Ambler. Agile/lean documentation: Strategies for agile software development. http://www.agilemodeling.com/essays/agileDocumentation.htm, accessed in Feburary 2012.

[9] M. Ames and M. Naaman. Why we tag: motivations for annotation in mobile and online media. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '07, pages 971–980, New York, NY, USA, 2007. ACM.

[10] C. Amrit. Coordination in software development: the problem of task allocation. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[11] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[12] A. Begel and R. DeLine. Codebook: Social networking over code. In *Proceedings of the 31st International Conference on Software Engineering - Companion Volume*, ICSE '09, pages 263 –266, Washington, DC, USA, 2009. IEEE Computer Society.

[13] A. Begel, R. DeLine, and T. Zimmermann. Social media for software engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 33–38, New York, NY, USA, 2010. ACM.

[14] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[15] A. Begel, K. Y. Phang, and T. Zimmermann. Whoselsthat: finding software engineers with codebook. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, FSE '10, pages 381–382, New York, NY, USA, 2010. ACM.

[16] A. Begel and T. Zimmermann. Keeping up with your friends: function foo, library bar.dll, and work item 24. In *Proceedings of the 1st Workshop on Web*

2.0 for Software Engineering, Web2SE '10, pages 20–23, New York, NY, USA, 2010. ACM.

[17] S. Bendifallah and W. Scacchi. Understanding software maintenance work. *IEEE Transactions on Software Engineering*, 13(3):311–323, 1987.

[18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, FSE '08, pages 308–318, New York, NY, USA, 2008. ACM.

[19] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 199–210, Washington, DC, USA, 2006. IEEE Computer Society.

[20] J. Bian, Y. Liu, E. Agichtein, and H. Zha. Finding the right facts in the crowd: factoid question answering over social media. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 467–476, New York, NY, USA, 2008. ACM.

[21] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: A visual dashboard for fostering awareness in software teams. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[22] S. Black, R. Harrison, and M. Baldwin. A survey of social media use in software systems development. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 1–5, New York, NY, USA, 2010. ACM.

[23] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and prune: a pragmatic approach to software product line implementation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '10, pages 333–336, New York, NY, USA, 2010. ACM.

[24] G. Bougie, J. Starke, M.-A. Storey, and D. M. German. Towards understanding twitter use in software engineering: preliminary findings, ongoing challenges and future questions. In *Proceedings of the 2nd International Workshop on Web 2.0*

for Software Engineering, Web2SE '11, pages 31–36, New York, NY, USA, 2011. ACM.

[25] F. P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[26] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[27] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: groupware for code inspection. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '90, pages 169–181, New York, NY, USA, 1990. ACM.

[28] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz. Enriching reverse engineering with annotations. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 660–674, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] S. L. Bryant, A. Forte, and A. Bruckman. Becoming wikipedian: transformation of participation in a collaborative online encyclopedia. In *Proceedings of the International ACM SIGGROUP conference on Supporting group work*, GROUP '05, pages 1–10, New York, NY, USA, 2005. ACM.

[30] F. Calefato, D. Gendarmi, and F. Lanubile. Embedding social networking information into jazz to foster group awareness within distributed teams. In *Proceedings of the 2nd International Workshop on Social Software Engineering and Applications*, SoSEA '09, pages 23–28, New York, NY, USA, 2009. ACM.

[31] F. Calefato, D. Gendarmi, and F. Lanubile. Investigating the use of tags in collaborative development environments: a replicated study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 24:1–24:9, New York, NY, USA, 2010. ACM.

[32] M. Cataldo, M. Bass, J. D. Herbsleb, and L. Bass. On coordination mechanisms in global software development. In *Proceedings of the International Conference on Global Software Engineering*, ICGSE '07, pages 71–80, Washington, DC, USA, 2007. IEEE Computer Society.

[33] L.-T. Cheng, M. Desmond, and M.-A. Storey. Presentations by programmers for programmers. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 788–792, Washington, DC, USA, 2007. IEEE Computer Society.

[34] J. Chudge and D. Fulton. Trust and co-operation in system development: applying responsibility modelling to the problem of changing requirements. *Software Engineering Journal*, 11(3):193–204, 1996.

[35] V. Clerc, E. de Vries, and P. Lago. Using wikis to support architectural knowledge management in global software development. In *Proceedings of the ICSE Workshop on Sharing and Reusing Architectural Knowledge*, SHARK '10, pages 37–43, New York, NY, USA, 2010. ACM.

[36] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the Symposium on Software Visualization*, SoftVis '03, pages 77–ff, New York, NY, USA, 2003. ACM.

[37] J. M. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1998.

[38] F. F. Correia, A. Aguiar, H. S. Ferreira, and N. Flores. Patterns for consistent software documentation. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, PLoP '09, pages 12:1–12:7, New York, NY, USA, 2010. ACM.

[39] F. F. Correia, H. S. Ferreira, N. Flores, and A. Aguiar. Incremental knowledge acquisition in software development using a weakly-typed wiki. In *Proceedings of the International Symposium on Wikis and Open Collaboration*, WikiSym '09, pages 31:1–31:2, New York, NY, USA, 2009. ACM.

[40] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[41] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries. Moving into a new software project landscape. In *Proceedings of the 32nd*

International Conference on Software Engineering - Volume 1, ICSE '10, pages 275–284, New York, NY, USA, 2010. ACM.

[42] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, FSE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[43] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 229–238, Washington, DC, USA, 2006. IEEE Computer Society.

[44] D. E. Damian and D. Zowghi. The impact of stakeholders? geographical distribution on managing requirements in a multi-site organization. In *Proceedings of the 10th International Conference on Requirements Engineering*, RE '02, pages 319–330, Washington, DC, USA, 2002. IEEE Computer Society.

[45] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: Software source code as a social and technical artifact. In *Proceedings of the International Conference on Supporting Group Work*, GROUP '05, pages 197–206, New York, NY, USA, 2005. ACM.

[46] C. R. B. de Souza, D. Redmiles, and P. Dourish. "Breaking the code", Moving between private and public work in collaborative software development. In *Proceedings of the International Conference on Supporting Group Work*, GROUP '03, pages 105–114, New York, NY, USA, 2003. ACM.

[47] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd International Conference on Design of Communication*, SIGDOC '05, pages 68–75, New York, NY, USA, 2005. ACM.

[48] B. Decker, E. Ras, J. Rech, P. Jaubert, and M. Rieth. Wiki-based stakeholder participation in requirements engineering. *IEEE Software*, 24(2):28–35, 2007.

[49] B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht. Self-organized reuse of software engineering knowledge supported by semantic wikis. In *Proceedings*

of the Workshop on Semantic Web Enabled Software Engineering, SWESE '05, 2005.

[50] U. Dekel and J. D. Herbsleb. Pushing relevant artifact annotations in collaborative software development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '08, pages 1–4, New York, NY, USA, 2008. ACM.

[51] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA, 2009. IEEE Computer Society.

[52] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[53] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer-Verlag, London, UK, 2008.

[54] W. Eckerson. *Performance dashboards: measuring, monitoring, and managing your business.* Wiley, 2005.

[55] L. Efimova and J. Grudin. Crossing boundaries: A case study of employee blogging. In *Proceedings of the 40th Hawaii International Conference on System Sciences*, HICSS '07, page 86, Washington, DC, USA, 2007. IEEE Computer Society.

[56] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft - A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[57] J. B. Ellis, S. Wahid, C. Danis, and W. A. Kellogg. Task and social visualization in software development: Evaluation of a prototype. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '07, pages 577–586, New York, NY, USA, 2007. ACM.

[58] T. Erickson and W. A. Kellogg. Social translucence: An approach to designing systems that support social processes. *ACM Transactions on Computer-Human Interaction*, 7(1):59–83, 2000.

[59] S. Faraj and L. Sproull. Coordinating expertise in software development teams. *Management Science*, 46(12):1554–1568, 2000.

[60] D. Ferreira and A. R. da Silva. Wiki supported collaborative requirements engineering. In *Proceedings of the Workshop on Wikis for Software Engineering*, Wiki4SE '08, 2008.

[61] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data: Research articles. *Journal of Software Maintenance and Evolution*, 16(6):385–403, 2004.

[62] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Control*, 17(4):367–394, 2009.

[63] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the Symposium on Document Engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.

[64] T. Fritz. Staying aware of relevant feeds in context. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 523–524, New York, NY, USA, 2010. ACM.

[65] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[66] T. Fritz and G. C. Murphy. Determining relevancy: how software developers determine relevant information in feeds. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '11, pages 1827–1830, New York, NY, USA, 2011. ACM.

[67] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society.

[68] R. Frost. Jazz and the eclipse way of collaboration. *IEEE Software*, 24(6):114–117, 2007.

[69] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[70] H. C. Gall and M. Lanza. Software evolution: analysis and visualization. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 1055–1056, New York, NY, USA, 2006. ACM.

[71] M. Geisser, T. Hildenbr, A. Korthaus, and S. Seedorf. New applications for wikis in software engineering. In *Proceedings of the PRIMIUM Subconference at the Multikonferenz Wirtschaftsinformatik (MKWI)*, PRIMIUM 2008, pages 145–160, Bonn, Germany, 2008. Gesellschaft für Informatik.

[72] E. M. Gerson and S. L. Star. Analyzing due process in the workplace. *ACM Transactions on Information Systems*, 4(3):257–270, 1986.

[73] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine, Piscataway, NJ, USA, 1967.

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

[75] R. E. Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996.

[76] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '04, pages 72–81, New York, NY, USA, 2004. ACM.

[77] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen. Collective code bookmarks for program comprehension. In *Proceedings of the 19th International Conference on Program Comprehension*, ICPC '11, pages 101–110, Washington, DC, USA, 2011. IEEE Computer Society.

[78] A. Guzzi, M. Pinzger, and A. van Deursen. Combining micro-blogging and ide interactions to support developers in their quests. In *Proceedings of the Interna-*

tional Conference on Software Maintenance, ICSM '10, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[79] Z. Gyöngyi, G. Koutrika, J. Pedersen, and H. Garcia-Molina. Questioning yahoo! answers. Technical Report 2007-35, Stanford InfoLab, 2007.

[80] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '06, pages 39–48, New York, NY, USA, 2006. ACM.

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

[82] A. Hassan and R. Holt. Using development history sticky notes to understand software architecture. In *Proceedings of the 12th International Workshop on Program Comprehension*, IWPC '04, pages 183 – 192, Washington, DC, USA, 2004. IEEE Computer Society.

[83] T. Hattori. Wikigramming: a wiki-based training environment for programming. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, Web2SE '11, pages 7–12, New York, NY, USA, 2011. ACM.

[84] J. D. Herbsleb, D. L. Atkins, D. G. Boyer, M. Handel, and T. A. Finholt. Introducing instant messaging and chat in the workplace. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '02, pages 171–178, New York, NY, USA, 2002. ACM.

[85] J. D. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 85–95, New York, NY, USA, 1999. ACM.

[86] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 81–90, Washington, DC, USA, 2001. IEEE Computer Society.

[87] J. D. Herbsleb and D. Moitra. Guest editors' introduction: Global software development. *IEEE Software*, 18(2):16–20, 2001.

[88] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the International working Conference on Mining Software Repositories*, MSR '08, pages 145–148, New York, NY, USA, 2008. ACM.

[89] P. Heymann, A. Paepcke, and H. Garcia-Molina. Tagging human knowledge. In *Proceedings of the International Conference on Web Search and Data Mining*, WSDM '10, pages 51–60, New York, NY, USA, 2010. ACM.

[90] R. Hoda, J. Noble, and S. Marshall. Organizing self-organizing teams. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ICSE '10, pages 285–294, New York, NY, USA, 2010. ACM.

[91] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th Symposium on User Interface Software and Technology*, UIST '07, pages 13–22, New York, NY, USA, 2007. ACM.

[92] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of the International working Conference on Mining Software Repositories*, MSR '08, pages 23–26, New York, NY, USA, 2008. ACM.

[93] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[94] R. Holmes and R. J. Walker. A newbie's guide to eclipse apis. In *Proceedings of the International working Conference on Mining Software Repositories*, MSR '08, pages 149–152, New York, NY, USA, 2008. ACM.

[95] J. Huh, L. Jones, T. Erickson, W. A. Kellogg, R. K. E. Bellamy, and J. C. Thomas. Blogcentral: the role of internal blogs at work. In *Proceedings of the Conference on Human Factors in Computing Systems - extended abstracts*, CHI '07, pages 2447–2452, New York, NY, USA, 2007. ACM.

[96] M. R. Jakobsen, R. Fernandez, M. Czerwinski, K. Inkpen, O. Kulyk, and G. G. Robertson. WIPDash: Work item and people dashboard for software development teams. In *Proceedings of the 12th International Conference on*

Human-Computer Interaction, INTERACT '09, pages 791–804, Berlin, Heidelberg, 2009. Springer-Verlag.

[97] L. Jen and Y. Lee. Working group. IEEE recommended practice for architectural description of software-intensive systems. *IEEE Architecture*, 2000.

[98] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.

[99] A. M. Kaplan and M. Haenlein. Users of the world, unite! The challenges and opportunities of social media. *Business Horizons*, 53(1):59–68, 2010.

[100] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *ECOOP'97 Object-Oriented Programming*, volume 1241, chapter Aspect-oriented programming, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[101] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[102] S. Komi-Sirviö, A. Mäntyniemi, and V. Seppänen. Toward a practical solution for capturing knowledge for software projects. *IEEE Software*, 19(3):60–62, 2002.

[103] R. E. Kraut and L. A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, 1995.

[104] F. Lanubile. Collaboration in distributed software development. In A. Lucia and F. Ferrucci, editors, *Software Engineering*, pages 174–193. Springer-Verlag, Berlin, Heidelberg, 2009.

[105] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno. Collaboration tools for global software engineering. *IEEE Software*, 27(2):52–55, 2010.

[106] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Languages et Modeles a Objets*, LMO '02, pages 135–149, London, UK, 2002. Hermes Publications.

[107] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Proceedings of the 2nd Workshop on the Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[108] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[109] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[110] S. Letovsky. Cognitive processes in program comprehension. In *Proceedings of the 1st Workshop on empirical Studies of Programmers*, ESP '86, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[111] B. Leuf and W. Cunningham. *The Wiki way: quick Collaboration on the Web.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[112] P. Liang, P. Avgeriou, and V. Clerc. Requirements reasoning for distributed requirements analysis using semantic wiki. In *Proceedings of the International Conference on Global Software Engineering*, ICGSE '09, pages 388–393, Washington, DC, USA, 2009. IEEE Computer Society.

[113] Y. S. Lincoln and E. G. Guba. *Naturalistic inquiry.* Sage Publications, Inc., Thousand Oaks, CA, USA, 1985.

[114] E. Lindqvist, B. Lundell, and B. Lings. Distributed development in an intra-national, intra-organisational context: an experience report. In *Proceedings of the International Workshop on Global Software Development for the Practitioner*, GSD '06, pages 80–86, New York, NY, USA, 2006. ACM.

[115] S. Lohmann, S. Dietzold, P. Heim, and N. Heino. A web platform for social requirements engineering. In *Proceedings of Software Engineering*, SE '09, pages 309–316, Bonn, Germany, 2009. Gesellschaft für Informatik.

[116] P. Louridas. Using wikis in software development. *IEEE Software*, 23(2):88–91, 2006.

[117] W. Maalej and H.-J. Happel. From work to word: How do software developers describe their work? In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR '09, pages 121–130, Washington, DC, USA, 2009. IEEE Computer Society.

[118] W. Maalej, D. Panagiotou, and H.-J. Happel. Towards effective management of software knowledge exploiting the semantic wiki paradigm. In *Proceedings of Software Engineering*, SE '08, pages 183–197, Bonn, Germany, 2008. Gesellschaft für Informatik.

[119] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '02, page 32, Washington, DC, USA, 2002. IEEE Computer Society.

[120] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '11, pages 2857–2866, New York, NY, USA, 2011. ACM.

[121] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the Conference on Programming language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[122] C. Marlow, M. Naaman, D. Boyd, and M. Davis. Ht06, tagging paper, taxonomy, flickr, academic article, to read. In *Proceedings of the 17th Conference on Hypertext and Hypermedia*, HYPERTEXT '06, pages 31–40, New York, NY, USA, 2006. ACM.

[123] L. Menand. *Pragmatism: A Reader*. Vintage Books, New York, NY, USA, 1997.

[124] A. Meneely, P. Rotella, and L. Williams. Does adding manpower also affect quality? an empirical, longitudinal analysis. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 81–90, New York, NY, USA, 2011. ACM.

[125] P. Mi and W. Scacchi. Modeling articulation work in software engineering processes. In *Proceedings of the 1st International Conference on the Software Process*, ICSP '91, pages 188–201, Washington, DC, USA, 1991. IEEE Computer Society.

[126] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 263–272, New York, NY, USA, 2000. ACM.

[127] A. Mockus and J. Herbsleb. Challenges of global software development. In *Proceedings of the 7th International Symposium on Software Metrics*, METRICS '01, pages 182–184, Washington, DC, USA, 2001. IEEE Computer Society.

[128] S. Nerur and V. Balijepally. Theoretical reflections on agile development methodologies. *Communications of the ACM*, 50(3):79–83, 2007.

[129] T. O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. http://oreilly.com/web2/archive/what-is-web-20.html, accessed in February 2012.

[130] H. Ossher, D. Amid, A. Anaby-Tavor, R. Bellamy, M. Callery, M. Desmond, J. De Vries, A. Fisher, S. Krasikov, I. Simmonds, and C. Swart. Using tagging to identify and organize concerns during pre-requirements analysis. In *Proceedings of the ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, EA '09, pages 25–30, Washington, DC, USA, 2009. IEEE Computer Society.

[131] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA, 2000. ACM.

[132] D. Pagano and W. Maalej. How do developers blog?: an exploratory study. In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR '11, pages 123–132, New York, NY, USA, 2011. ACM.

[133] S. Park and F. Maurer. The role of blogging in generating a software product vision. In *Proceedings of the International Workshop on Cooperative and Human*

Aspects on Software Engineering, CHASE '09, pages 74–77, Washington, DC, USA, 2009. IEEE Computer Society.

[134] D. L. Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer-Verlag, Berlin, Heidelberg, 2011.

[135] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[136] C. Parnin and C. Treude. Measuring api documentation on the web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, Web2SE '11, pages 25–30, New York, NY, USA, 2011. ACM.

[137] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA, 2000. ACM.

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

[139] A. J. Phuwanartnurak and D. G. Hendry. Understanding information sharing in software development through wiki log analysis. In *Proceedings of the International Symposium on Wikis and Open Collaboration*, WikiSym '09, pages 35:1–35:2, New York, NY, USA, 2009. ACM.

[140] A. L. Powell, J. C. French, and J. C. Knight. A systematic approach to creating and maintaining software documentation. In *Proceedings of the Symposium on Applied Computing*, SAC '96, pages 201–208, New York, NY, USA, 1996. ACM.

[141] J. Rech, C. Bogner, and V. Haas. Using wikis to tackle reuse in software projects. *IEEE Software*, 24(6):99–104, 2007.

[142] W. Reinhardt. Communication is the key - support durable knowledge sharing in software engineering by microblogging. In *Proceedings of the Workshop on Software Engineering within Social Software Environments*, SENSE '09, 2009.

[143] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[144] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416, New York, NY, USA, 2002. ACM.

[145] M. P. Robillard and F. Weigand-Warr. Concernmapper: simple view-based separation of scattered concerns. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 65–69, New York, NY, USA, 2005. ACM.

[146] P. N. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, 1999.

[147] V. Robu, H. Halpin, and H. Shepherd. Emergence of consensus and shared vocabularies in collaborative tagging systems. *ACM Transactions on Web*, 3(4):1–34, 2009.

[148] K. Rönkkö, Y. Dittrich, and D. Randall. When plans do not work out: How plans are used in software development projects. *Computer Supported Cooperative Work*, 14(5):433–468, 2005.

[149] J. Rowley. What is knowledge management. *Library Management*, 20(8):416–419, 1999.

[150] I. Rus and M. Lindvall. Guest editors' introduction: Knowledge management in software engineering. *IEEE Software*, 19(3):26–38, 2002.

[151] F. V. Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th International Conference on Software Maintenance*, ICSM '04, pages 328–337, Washington, DC, USA, 2004. IEEE Computer Society.

[152] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.

[153] A. Sarma and A. van der Hoek. Towards awareness in the large. In *Proceedings of the International Conference on Global Software Engineering*, ICGSE '06, pages 127–131, Washington, DC, USA, 2006. IEEE Computer Society.

[154] S. Sen, S. K. Lam, A. M. Rashid, D. Cosley, D. Frankowski, J. Osterhouse, F. M. Harper, and J. Riedl. tagging, communities, vocabulary, evolution. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '06, pages 181–190, New York, NY, USA, 2006. ACM.

[155] B. Sengupta, S. Chandra, and V. Sinha. A research agenda for distributed software development. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 731–740, New York, NY, USA, 2006. ACM.

[156] N. Seyff, F. Graf, and N. Maiden. End-user requirements blogging with irequire. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 285–288, New York, NY, USA, 2010. ACM.

[157] C. Shah and J. Pomerantz. Evaluating and predicting answer quality in community QA. In *Proceedings of the 33rd International Conference on Research and Development in information retrieval*, SIGIR '10, pages 411–418, New York, NY, USA, 2010. ACM.

[158] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*, FSE '06, pages 23–34, New York, NY, USA, 2006. ACM.

[159] K. Starbird and L. Palen. (how) will the revolution be retweeted?: information diffusion and the 2011 egyptian uprising. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '12, pages 7–16, New York, NY, USA, 2012. ACM.

[160] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW '06, pages 195–198, New York, NY, USA, 2006. ACM.

[161] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 251–260, New York, NY, USA, 2008. ACM.

[162] M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, and M. Muller. How software developers use tagging to support reminding and refinding. *IEEE Transactions on Software Engineering*, 35(4):470–483, 2009.

[163] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 359–364, New York, NY, USA, 2010. ACM.

[164] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.

[165] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: improving api documentation using usage information. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems - extended abstracts*, CHI '09, pages 4429–4434, New York, NY, USA, 2009. ACM.

[166] B. Tansey and E. Stroulia. Annoki: a mediawiki-based collaboration platform. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 31–36, New York, NY, USA, 2010. ACM.

[167] A. Telea and L. Voinea. Interactive visual mechanisms for exploring source code evolution. In *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, page 17, Washington, DC, USA, 2005. IEEE Computer Society.

[168] C. Treude. The role of emergent knowledge structures in collaborative software development. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 389–392, New York, NY, USA, 2010. ACM.

[169] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. ACM.

[170] C. Treude, F. Filho, B. Cleary, and M.-A. Storey. Programming in a socially networked world: the evolution of the social programmer. The Future of Collaborative Software Development (FutureCSD), 2012.

[171] C. Treude, P. Gorman, L. Grammel, and M.-A. Storey. Workitemexplorer: Visualizing software development tasks using an interactive exploration environment. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1416–1419, Washington, DC, USA, 2012. IEEE Computer Society.

[172] C. Treude and M.-A. Storey. Concernlines: A timeline view of co-occurring concerns. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 575–578, Washington, DC, USA, 2009. IEEE Computer Society.

[173] C. Treude and M.-A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 12–22, Washington, DC, USA, 2009. IEEE Computer Society.

[174] C. Treude and M.-A. Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ICSE '10, pages 365–374, New York, NY, USA, 2010. ACM.

[175] C. Treude and M.-A. Storey. Bridging lightweight and heavyweight task organization: the role of tags in adopting new task categories. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 231–234, New York, NY, USA, 2010. ACM.

[176] C. Treude and M.-A. Storey. The implications of how we tag software artifacts: exploring different schemata and metadata for tags. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 12–13, New York, NY, USA, 2010. ACM.

[177] C. Treude and M.-A. Storey. Effective communication of software development knowledge through community portals. In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 91–101, New York, NY, USA, 2011. ACM.

[178] C. Treude and M.-A. Storey. Work item tagging: Communicating concerns in collaborative software development. *IEEE Transactions on Software Engineering*, 38(1):19–34, 2012.

[179] C. Treude, M.-A. Storey, K. Ehrlich, and A. van Deursen. Web2se: First workshop on web 2.0 for software engineering. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 457–458, New York, NY, USA, 2010. ACM.

[180] C. Treude, M.-A. Storey, K. Ehrlich, and A. van Deursen. Workshop report from web2se: first workshop on web 2.0 for software engineering. *SIGSOFT Software Engineering Notes*, 35(5):45–50, 2010.

[181] C. Treude, M.-A. Storey, A. van Deursen, A. Begel, and S. Black. Second international workshop on web 2.0 for software engineering (web2se 2011). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1222–1223, New York, NY, USA, 2011. ACM.

[182] C. Treude, M.-A. Storey, A. van Deursen, A. Begel, and S. Black. Workshop report from web2se 2011: 2nd international workshop on web 2.0 for software engineering. *SIGSOFT Software Engineering Notes*, 36(5):24–29, 2011.

[183] C. Treude, M.-A. Storey, and J. Weber. Empirical studies on collaboration in software development: A systematic literature review. Technical report, Technical Report DCS-331-IR, Department of Computer Science, University of Victoria, 2009.

[184] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: a knowledgeable, browser-based ide. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ICSE '10, pages 203–206, New York, NY, USA, 2010. ACM.

[185] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[186] G. Venolia. Textual allusions to artifacts in software-related repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR '06, pages 151–154, New York, NY, USA, 2006. ACM.

[187] M. Visconti and C. R. Cook. An overview of industrial software documentation practice. In *Proceedings of the 12th International Conference of the Chilean Computer Science Society*, SCCC '02, page 179, Washington, DC, USA, 2002. IEEE Computer Society.

[188] J. Voas. Faster, better, and cheaper. *IEEE Software*, 18(3):96–97, 2001.

[189] L. Voinea and A. Telea. Mining software repositories with cvsgrab. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR '06, pages 167–168, New York, NY, USA, 2006. ACM.

[190] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: Visualization of code evolution. In *Proceedings of the Symposium on Software Visualization*, SoftVis '05, pages 47–56, New York, NY, USA, 2005. ACM.

[191] D. Woit and K. Bell. Student communication challenges in distributed software engineering environments. In *Proceedings of the 10th Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 286–290, New York, NY, USA, 2005. ACM.

[192] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, IWPSE '04, pages 57–66, Washington, DC, USA, 2004. IEEE Computer Society.

[193] W. Xiao, C. Chi, and M. Yang. On-line collaborative software development via wiki. In *Proceedings of the International Symposium on Wikis and Open Collaboration*, WikiSym '07, pages 177–183, New York, NY, USA, 2007. ACM.

Appendices

Appendix A

Interview and Survey Questions

In this appendix, I present the interview and survey questions used in the empirical studies. All interviews were semi-structured allowing for follow-up questions and clarifications.

A.1 Tagging in IBM's Jazz

I used the following questions to guide the interviews with developers using IBM's Jazz regarding their use of work item tags. The corresponding findings are presented in Section 3.1.

- For how long have you been with the team?
- What are you working on?
- What is your role?
- Do you tag work items?
- Why do you tag work items?
- How many work items do you tag?
- Do you use tags for work items?
- How do you use tags for work items?
- Why do you use tags for work items?

- Not all work items are tagged. Is that a problem?
- How would you usually get your work items?
- How do you choose tag names?
- Are there any kind of conventions within the team on tags?
- Do you introduce new tags?
- Is there any other meta information that would be useful in tags?
- Are there different kinds of tags?
- Are tags used to overcome shortcomings of the UI?
- Tags are usually used to describe the content of what the tag is applied to. Is that the same with tags for work items?
- For whom do you create tags?
- Is there a need for private tags?
- Do you use tags that others apply to work items?
- Do you ever use tags to communicate to other developers?
- Are there conflicts related to tagging others' work items?
- Is it helpful to see who created tags?
- Do you remove tags from work items?
- Do your tags ever get removed by others?
- When do you tag (right after work item creation)?
- In what context do you tag (meeting etc.)?
- Has your tagging behaviour changed over time?
- Does your tag cloud mirror your area of work?
- What do these tags mean?

- Outside of Jazz, do you use any tagging applications?
- How can tags for work items be improved?

A.2 Dashboards and Feeds in IBM's Jazz

I used the following questions in the survey with developers using IBM's Jazz regarding their use of dashboards and feeds. The corresponding findings are presented in Section 4.1.

- When did you start to develop software using IBM's Jazz? *Select One*
  - in 2009
  - in 2008
  - in 2007
  - in 2006
  - before 2006
- On how many different teams are you an active participant? *Small Text Field*
- Which of these roles do you currently hold? *Multiple Choice*
  - Contributor
  - Component Lead
  - Integration Stream Admin
  - Integration Build Meister
  - Release Manager
  - Release Engineer
  - Dashboard Admin
  - PMC
  - Development Manager
  - Project Admin
  - Other: *Small Text Field*

- Out of 100%, how many percent of your time do you approximately spend on project manager activities? *Small Text Field*
- Out of 100%, how many percent of your time do you approximately spend on developer activities? *Small Text Field*
- Out of 100%, how many percent of your time do you approximately spend on other activities? *Small Text Field*
- When was the last time you looked at your Team Central view in Team Concert? Please check the first answer that applies. *Select One*
  - Within the last 10 minutes
  - Within the last hour
  - Within the last 24 hours
  - Within the last week
  - I don't remember
  - I don't use Team Central
- What question were you trying to answer? *Large Text Field*
- Were you able to answer that question using Team Central? *Select One*
  - Yes
  - No
- On average, how often do you look at Team Central? *Select One*
  - Several times each hour
  - About once an hour
  - Several times each day
  - About once a day
  - Several times each week
  - About once a week
  - Rarely
  - Never

- Is the extent of your Team Central use constant across different project phases? *Select One*
  - Yes
  - No
- If no, please explain: *Large Text Field*
- What is your main reason for using Team Central? *Large Text Field*
- Which of the following sections are visible in your Team Central? *Multiple Choice*
  - My Open Work Items
  - Event Log
  - Team Load
  - Build
  - Recent Work
  - New Unassigned Work Items
  - News
  - Queries
  - Other: *Small Text Field*
- Please explain in a few sentences why you use these sections. *Large Text Field*
- What information is missing in Team Central? *Large Text Field*
- What tool enhancements would be useful for Team Central? *Large Text Field*
- Do you use Feeds in Jazz outside of Team Central? *Select One*
  - Yes
  - No
- If no, please specify: *Large Text Field*
- Which Feeds do you regularly look at? *Multiple Choice*

  - Build Events for My Teams
  - My Work Item Changes
  - My Teams in Project Area
  - Other: *Small Text Field*

- On average, how often do you look at Feeds? *Select One*
  - Several times each hour
  - About once an hour
  - Several times each day
  - About once a day
  - Several times each week
  - About once a week
  - Rarely
  - Never

- Is the extent of your use of Feeds constant across different project phases? *Select One*
  - Yes
  - No

- If no, please explain: *Large Text Field*

- What is your main reason for using Feeds? *Large Text Field*

- Do you care about how your work is reflected in other developers' Feeds? *Select One*
  - Yes
  - No

- If yes, please explain: *Large Text Field*

- Are Feeds distracting? *Select One*
  - Yes

  - No

- If yes, please explain: *Large Text Field*
- When was the last time you looked at a Dashboard in Jazz? Please check the first answer that applies. *Select One*
  - Within the last 10 minutes
  - Within the last hour
  - Within the last 24 hours
  - Within the last week
  - I don't remember
  - I don't use Dashboards
- What question were you trying to answer? *Large Text Field*
- Were you able to answer that question using Dashboards? *Select One*
  - Yes
  - No
- On average, how often do you look at Dashboards? *Select One*
  - Several times each hour
  - About once an hour
  - Several times each day
  - About once a day
  - Several times each week
  - About once a week
  - Rarely
  - Never
- Is the extent of your Dashboard use constant across different project phases? *Select One*
  - Yes

  - No

- If no, please explain: *Large Text Field*
- What is your main reason for using Dashboards? *Large Text Field*
- Which of these Dashboards do you look at the most? *Select One*
  - Project Area
  - Team Area
  - Contributor
  - I don't look at Dashboards
- How many different Dashboards have you edited in the past? *Small Text Field*
- How many of these Dashboards are shared? *Small Text Field*
- Have you ever edited a Team Area Dashboard? *Select One*
  - Yes
  - No
- Have you ever edited a Project Area Dashboard? *Select One*
  - Yes
  - No
- Do you use Dashboards to navigate to work items? *Select One*
  - Yes
  - No
- Which of the following viewlet types do you regularly look at? *Multiple Choice*
  - Feeds
  - Work Item Statistics (e.g., bar chart, tag cloud)
  - Work Items (result of a single work item query)
  - Bookmarks
  - HTML (static)

  - Work Item Queries (bookmarks to several queries)
  - About Me
  - Other: *Small Text Field*

- Please explain in a few sentences why you use these viewlet types. *Large Text Field*
- Which of the following visualizations do you regularly look at? *Multiple Choice*
  - Vertical Bar Chart
  - Horizontal Bar Chart
  - Pie Chart
  - Tag Cloud
  - Table
- Please explain in a few sentences why you use these visualizations. *Large Text Field*
- Do you care about how your work is reflected in Dashboards? *Select One*
  - Yes
  - No
- If yes, please explain: *Large Text Field*
- Does the information displayed in Dashboards alter your work practices? *Select One*
  - Yes
  - No
- If yes, please explain: *Large Text Field*
- Are Dashboards distracting? *Select One*
  - Yes
  - No
- If yes, please explain: *Large Text Field*

- What information is missing in Dashboards? *Large Text Field*
- What tool enhancements would be useful for Dashboards? *Large Text Field*

A.3 Documentation of IBM's Jazz

I used the following questions to guide the interviews with developers of IBM's Jazz client Rational Team Concert (RTC) regarding their use of the community portal jazz.net. The corresponding findings are presented in Section 6.1.

- What is your role / different roles?
- For how long have you been in that position?
- How do you use Jazz?
- How do you learn about Jazz features / functionalities?
- Is that different from how you learn about other products?
- When was the last time you looked at jazz.net?
- What question were you trying to answer?
- How did jazz.net answer that question? What kind of artifact?
- How did you find information?
- Do you use jazz.net often?
- Why do / don't you use jazz.net?
- How do you use jazz.net?
- How do you use wikis / forums / mailing lists / articles / online help / podcasts / presentations / tech tips / videos / blogs?
- When and how frequently do you look at jazz.net?
- What kind of question do you ask using jazz.net?
- Could this question be answered without jazz.net? If so, how?

- What difference does this information make?
- For whom is jazz.net useful (team-internal vs. customers)?
- What are the difficulties and frustrations of using jazz.net?
- Were you always able to find what you were looking for?
- How are portals for other products that you use structured and how does that compare to jazz.net?
- Have you ever contributed to jazz.net? If so, how?
- What / Who triggered the contribution?
- Are there conventions within the team for contributions to jazz.net?
- Have other team members helped you with it / commented on it?
- Have you received feedback on your contribution? From whom? How?
- What sources did you use for your contribution?
- Why did you use this kind of artifact and not another one (video / article / wiki etc)?
- For whom did you write?
- What did you do to ensure people would be able to find it?
- Do you know if people look at it?
- Have you changed it afterwards?
- Have you ever deleted something on jazz.net?
- What are the different kinds of artifacts on jazz.net used for?
- Who uses which kind of artifact?
- How well does jazz.net inform customers and other developers?
- Which artifacts are effective at communicating information?
- What information is missing on jazz.net / are there gaps?

- How could that information be added?
- How could processes for documentation be improved?
- How could tool support be improved?

Appendix B

WorkItemExplorer Evaluation

This appendix presents additional data related to the evaluation of WorkItemExplorer (see Section 4.2).

B.1 Developers' Questions that WorkItemExplorer Can Answer

To evaluate how well WorkItemExplorer can answer developers' questions, we used the questions that Fritz and Murphy [65] identified as questions that developers frequently ask as a benchmark. The following lists all 78 questions they identified. In bold denoted are the questions that developers can currently answer using WorkItemExplorer, and for the remaining questions, it is denoted what additional data the tool would need to process in order to be able to answer those questions.

1. **Who is working on what?**
2. **What are they [coworkers] working on right now?**
3. **What have other people been working on?**
4. **How much work [have] people done?**
5. Who changed this [code], focused on person? *source code*
6. Who to assign a code review to? / Who has the knowledge to do the code review? *source code*

7. **What [have] people done lately?**
8. **Who is working on what at the moment?**
9. **What has [a particular team member] been doing?**
10. **What have people been working on?**
11. Which code reviews have been assigned to which person? *reviews*
12. Who to assign a code review to? / Who has time for a code review? *reviews*
13. What is the evolution of the code? *source code*
14. Why were they [these changes] introduced? *source code*
15. Who made a particular change and why? *source code*
16. What classes has my team been working on? *source code*
17. What are the changes on newly resolved work items related to me? *source code*
18. Who is working on the same classes as I am and for which work item? *source code*
19. Who changed this [code], focused on code? *source code*
20. What is the whole history of this file? *source code*
21. What has been happening on [this] class? *source code*
22. What [have] people changed lately? *source code*
23. What changes have been made and why? *source code*
24. What has changed between two builds [and] who has changed it? *source code*
25. Who has made changes to my classes ?*source code*
26. Who is using that API [that I am about to change]? *source code*
27. Who created the API [that I am about to change]? *source code*
28. Who owns this piece of code? / Who modified it the latest? *source code*

29. Who owns this piece of code? / Who modified it most? *source code*
30. Who to talk to if you have to work with packages you haven't worked with? *source code*
31. How much has changed [in the project code]? *source code*
32. [Is anyone] intending to commit anything to that class? *source code*
33. Where have changes been made related to you? *source code*
34. Who is responsible for this code? (Who made the latest change?) *source code*
35. Which team is responsible for this code? (Who [has] made most changes to the code?) *source code*
36. What classes have been changed? *source code*
37. [Which] API has changed (to see which methods are not supported any more)? *source code*
38. What's the most popular class? [Which class has been changed most?] *source code*
39. Which other code that I worked on uses this code pattern / utility function? *source code*
40. Which code has recently changed that is related to me? *source code*
41. How do recently delivered changes affect changes that I am working on? *source code*
42. What code is related to a change? *source code*
43. Where has code been changing [this week]? *source code*
44. Which classes have been changed between two builds? *source code*
45. What is going on [in a package]? *source code*
46. [Which] changes [have been made] between these days or after this day? *source code*

47. What classes in this component were modified since version [...]? *source code*
48. What is the recent activity on a plan item? *history*
49. Which features and functions have been changing? *history*
50. Has progress been made on blockers (blocking work items) in your milestone? *history*
51. **Which work items/plan items are most active?**
52. **How active is the [plan item]? [How many comments were made on related work items?]**
53. **Are there any new comments on interesting work items?**
54. **What work item has recently changed that is related to me?**
55. [What are the] emails related to line items and defects that are features? *email*
56. **What are the comments on newly resolved work items that are related to me?**
57. Is progress (changes) being made on plan items? *history*
58. What is the activity on a line item (feature)? *history*
59. What caused this build to break? (Which change caused the stack trace?) *builds*
60. What has caused this built to break? (look at stack trace and intersect with change sets) *builds*
61. Who caused this build to break? (Who owns the broken tests?) *builds*
62. Who changed the test case most recently that caused the build to fail? *builds*
63. Which changes caused the tests to fail and thus the build to break? *builds*
64. Who owns a test case? (Who resolved the last work item that fixed the test case?) *tests*
65. Who is responsible for a failing test case? (stack trace) *tests*

66. How do test cases relate to work items? *tests*

67. How do test cases relate to packages/classes? *tests*

68. Which API has changed (check on web site)? *feeds*

69. [Is an entry] in newsgroup forum addressed to me because of the class mentioned? *forums*

70. **What is coming up next week [for my team]? [What is my team doing?]**

71. **What am I supposed to work on [plan on wiki]?**

72. **Who has to do what? (team activity)**

73. **How is the team organized?**

74. Who has made changes to [a] defect? *history*

75. **Who has made comments in defect?**

76. [What is] the collaboration tree around a feature? *history*

77. Which conversations in work items have I been mentioned in? *feeds*

78. **What are people commenting [on] all work items I am involved with?**

Appendix C

Key-Value Pairs on Google Code

This appendix presents additional data related to the study of task management on Google Code (see Section 5.1).

C.1 Most-Used Keys in Key-Value Pairs

Table C.1: Keys with the most instances across all projects

key	instances	distinct projects
`Priority`	83,752	987
`Type`	82,224	989
`Milestone`	14,489	194
`Component`	12,255	261
`ReportedBy`	11,076	1
`Version`	6,380	18
`OpSys`	2,692	215
`Subcomponent`	1,902	1
`Model`	996	1
`Module`	966	7

Across all 1,000 projects in the data, 166 different keys had been used. Table C.1 shows the ten most-used keys along with the information how many different projects used each key.

In the Android project, eight different keys had been used. They are shown in Table C.2.

Developers of CyanogenMod used seven different keys, as shown in Table C.3.

Table C.2: Keys with the most instances in Android

key	instances
`Type`	18,629
`Priority`	18,629
`ReportedBy`	11,076
`Component`	4,596
`Version`	3,190
`Subcomponent`	1,902
`Target`	393
`Restrict`	2

Table C.3: Keys with the most instances in CyanogenMod

key	instances
`Type`	3,741
`Priority`	3,713
`Version`	1,657
`Model`	996
`Framework`	19
`Audio`	14
`Network`	4