Studying the Impact of Developer Communication on the Quality and Evolution of a Software System

by

Nicolas Bettenburg

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Philosophical Doctor

Queen's University
Kingston, Ontario, Canada
March 2014

Abstract

Software development is a largely collaborative effort, of which the actual encoding of program logic in source code is a relatively small part. Software developers have to collaborate effectively and communicate with their peers in order to avoid coordination problems. To date, little is known how developer communication during software development activities impacts the quality and evolution of a software.

In this thesis, we present and evaluate tools and techniques to recover communication data from traces of the software development activities. Such data is recorded in software development repositories, such as version control systems, code review databases, issue report tracking systems, or email communication archives.

With this data, we study the impact of developer communication on the quality and evolution of the software through an in-depth investigation of the role of developer communication during software development activities.

Through multiple case-studies on a broad spectrum of open-source software projects, we find that communication between developers stands in a direct relationship to the quality of the software. Through the development and analysis of statistical models on how different aspects of communication such as content, participants, information flow, and timing relate to software quality, we observe that qualitative and quantitative aspects of developer communication can be used in these statistical models to explain software defects. Our findings demonstrate that our models based on developer communication explain software defects as well as state-of-the art models that are based on technical information such as code and process metrics, and that social information metrics are orthogonal to these traditional metrics, leading to a more complete and integrated view on software defects.

In addition, we find that communication between developers plays a important role in maintaining a healthy contribution management process, which is one of the key factors to the successful evolution of the software. Source code contributors who are part of the community surrounding open-source projects are available for limited times, and long communication times can lead to the loss of valuable contributions.

Our thesis illustrates that software development is an intricate and complex process that is strongly influence by the social interactions between the stakeholders involved in the development activities. A traditional view based solely on technical aspects of software development such as source code size and complexity, while valuable, limits our understanding of software development activities. The research presented in this thesis makes a strong case for the value of understanding these social and socio-technical aspects of development activities, and together with the tools and techniques presented in earlier chapters of this thesis consists of a first step towards gaining a more holistic view on software development activities.

Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

_____ _____

(Nicolas Bettenburg) (March, 17th 2014)

Acknowledgements

First, I would like express my sincere gratitude to my supervisor, Ahmed E. Hassan. Ahmed gave me the freedom and guidance to explore my academic and industrial research interests, without which this thesis would not have been possible. I would also like to thank my committee members, Jim Cordy, Nick Graham, and Rob DeLine. Through expert critique and suggestions, they helped shape this thesis, and the research ideas it is based upon.

Furthermore, I want to thank my collaborators and co-authors: Bram Adams, Emad Shihab, Meiyappan Nagappan, Stephen Thomas and Walid Ibrahim. Special thanks go out to Thanh Nguyen for showing me that there is a shower in the lab that provides just the right refreshment after a night of number-crunching. My dear fellows, together we faced this incredible journey through the intricacies and complexities of empirical studies and experimentation, qualitative and quantitative analysis, statistics, academic writing, and preparation of replication packages. I am eternally grateful to have been on that journey with you.

Queen's University's Software Analysis and Intelligence Laboratory (SAIL), is a fantastic melting pot ranging from undergraduate students with fresh ideas, over world-class visiting academics and practitioners, to a fantastic support team of outstanding post-doctoral researchers, who provided just the right mix of discussion, feedback, motivation, and peer-support. It is my sincere hope that future students will have the same opportunity to thrive in this academic environment that I was given during my time at SAIL.

I would like to thank my team lead and collaborator, Andrew Begel. During my time at Microsoft Research, Andrew opened my eyes to the importance and relevancy of the applicability of academic research in real world software development.

Finally, thanks to all of my friends and family, both here in Canada and far, back home in the Old World, who were the main driver that kept me going. Your friendship, love and continued support mean the wold to me.

Dedication

This thesis is dedicated to Hilde and, Rolf. Best. Parents. EVER!

Related Publications

The following publications are related to this thesis:

1. (Chapter 1) **Nicolas Bettenburg** and Ahmed E. Hassan: *Mining development repositories to study the impact of collaboration on software systems.* In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, p. 376-379. http://doi.acm.org/10.1145/2025113.2025165

2. (Chapter 2) **Nicolas Bettenburg** and Ahmed E. Hassan: *A Systematic Literature Review on the Relationships between Socio-Technical Information and Software Quality.* In Journal of Systems and Software. Under Submission. Elsevier, 2014.

3. (Chapter 3) **Nicolas Bettenburg**, Emad Shihab and Ahmed E. Hassan: *An empirical study on the risks of using off-the-shelf techniques for processing mailing list data.* In Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM 2009). IEEE Computer Society, Washington, DC, USA, p. 539-542. http://dx.doi.org/10.1109/ICSM.2009.5306383

4. (Chapter 4) **Nicolas Bettenburg**, Bram Adams, Ahmed E. Hassan, and Michel Smidt: *A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data.* In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC '11). IEEE Computer Society, Washington, DC, USA, p. 185-188. http://dx.doi.org/10.1109/ICPC.2011.36

5. (Chapter 5) **Nicolas Bettenburg**, Stephen W. Thomas, Ahmed E. Hassan: *Using fuzzy code search to link code fragments in discussions to source code.* In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2012). IEEE Computer Society, Washington, DC, USA, p. 319-328. http://doi.ieeecomputersociety.org/10.1109/CSMR.2012.39

6. (Chapter 6) **Nicolas Bettenburg**, Meiyappan Nagappan, Ahmed E. Hassan: *Towards Improving Statistical Modelling of Software Engineering Data: Think Locally, Act Globally!* In Journal of Empirical Software Engineering, Accepted November 27 2013.

   An earlier version of this work was published as **Nicolas Bettenburg**, Meiyappan Nagappan, Ahmed E. Hassan: *Think locally, act globally: Improving defect and effort prediction models*. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, IEEE Computer Society, Washington, DC, USA, p.60-69. http://dx.doi.org/10.1109/MSR.2012.6224300 and won the conference's Best Paper award.

7. (Chapter 7) **Nicolas Bettenburg**, Ahmed E. Hassan, Bram Adams, and Daniel M. German: *Management of community contributions.* In Journal of Empirical Software Engineering (2013): 1-38. http://dx.doi.org/10.1007/s10664-013-9284-6

The following publications were done during the research on this thesis and have inspired many of our research results, as well as provided a deeper understanding the intricacies of software quality.

1. **Nicolas Bettenburg** and Andrew Begel: *Deciphering the story of software development through frequent pattern mining.* In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1197-1200. http://dx.doi.org/10.1109/ICSE.2013.6606677

2. Walid M. Ibrahim, **Nicolas Bettenburg**, Bram Adams, Ahmed E. Hassan: *On the Relationship between Comment Update Practices and Software Bugs.* Journal of System and Software, Volume 85, Issue 10, October 2012, Pages 2293-2304. http://dx.doi.org/10.1016/j.jss.2011.09.019

3. Walid M. Ibrahim, **Nicolas Bettenburg**, Emad Shihab, Bram Adams, and Ahmed E. Hassan: *Should I contribute to this discussion?* In Proceedings of the 7th Working Conference on Mining Software Repositories (MSR 2010). Cape Town, South Africa. May 2-8, 2010. http://dx.doi.org/10.1109/MSR.2010.5463345

4. Emad Shihab, **Nicolas Bettenburg**, Bram Adams and Ahmed E. Hassan: *On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study.* In Proceedings of of the 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD) 2009, Tokyo, Japan, Nov. 19-20, 2009. http://dx.doi.org/10.1007/978-3-642-14888-0_9

5. Adrian Schroeter, **Nicolas Bettenburg**, and Rahul Premraj. *Do Stacktraces Help Developers Fix Bugs?*. In Proceedings of the 7th International Working Conference on Mining Software Repositories (MSR 2010). http://dx.doi.org/10.1109/MSR.2010.5463280

6. **Nicolas Bettenburg**, Meiyappan Nagappan and Ahmed E. Hassan: *Towards improving statistical modeling of software engineering data: think locally, act globally!*. In Journal of Empirical Software Engineering, Accepted January 2014. http://dx.doi.org/10.1007/s10664-013-9292-6

   An earlier version of this work was published as **Nicolas Bettenburg**, Meiyappan Nagappan and Ahmed E. Hassan: *Think locally, act globally: Improving defect and effort prediction models*. In Proceedings of the 9th IEEE Working Conference Mining Software Repositories (MSR 2012), pp.60-69. http://dx.doi.org/10.1109/MSR.2012.6224300 and won the conference's Best Paper award.

Contents

List of Tables

List of Figures

1 Introduction

Source code is the end-product of a variety of collaborative activities, carried out by the developers of a software. Lately, research has begun to understand that the intricacies of these activities, such as social networks, work dependencies, or daily routines, stand in a direct relation to the successful evolution of the software. Traces of these collaborative activities are captured throughout the development of the software and can be found in a large variety of repositories that developers use on a day-today basis, such as issue tracking systems, email communication archives, or version archives. In this dissertation, we present tools and methods to mine communication data, as an artifact describing the collaborative efforts of software developers. We use the extracted information to empirically study the relationship between communication and the evolution of the software. We study two manifestations of developer communication. First we study developer communication that is focussed on issue reports such as bug reports. Second, we study developer communication that is focussed on source code contributed by peer developers. The aim of our work is to feedback the insights gained from our empirical studies, to assist stakeholders in making future decisions and ultimately increasing software quality, as well as development efficiency.

Software development is a highly complex task. In order to cope with this complex task, development effort is often split across individuals or teams, who are responsible for one or more (less complex) concerns of the development effort [Baldwin1999; Parnas1972]. Managers often use high-level information about the software system to divide the development effort into work teams [Cataldo2008b]. However, such a division of the development effort increases the need for coordination and communication among developers [Kraut1995].

Melvin Conway in 1968 informally described this duality between modularization of the source code and the organization of developers responsible for creating that source code as Conway's Law. The source code of a software system stands in direct relation with the organizational structure of the development team, as design decisions require communication among the stakeholders making those decisions [Conway1968].

Based on that need for communication and coordination, it comes to no surprise that software developers typically spend a large fraction of their work day communicating with their co-workers [Begel2010]. For instance, Wu et al. report in an observational study on a large industrial software development company that engineers spent up to 2 hours each day communicating with their co-workers [Wu2003].

Recent research suggests that communication has a direct impact on the software development efforts. For instance, de Souza et al. provided evidence that communication among software developers is a critical factor for the success of software development efforts: information hiding led development teams to be unaware of other teams' work, resulting in coordination problems [Souza2004]. Similar findings for distributed software development have been reported by Bird et al. and Grinter et al. [Grinter1999; Bird2009a], who highlight communication as the most referenced problem in distributed software development [Bird2009a].

Lately, research in Empirical Software Engineering has begun to understand that the multitude of social and interpersonal interactions that go hand-in-hand with communication may also impact the software development process, as well as the software product itself. For instance, research has demonstrated that the forming of social networks and sub-communities [Wolf2009;

Bird2008], work dependencies [Cataldo2009b], or daily routines [Sliwerski2005] stand in a direct relation to the quality of the final software product [Cataldo2008b].

1.1 Thesis Hypothesis

Motivated by the empirical evidence provided in past research, this dissertation proposes to study the impact of communication between stakeholders in the software development process, such as developers and users, on the quality of the software. We believe that *the quality of a software system*, is impacted not only by the condition of the source code, but also by the collaborative activities of the stakeholders responsible for the development of the software system.
These beliefs led us to the formulation of our research hypothesis, which we state as follows:

The communication between software developers plays a key role in the quality and the evolution of the software.

Communication between stakeholders can materialize in a broad spectrum of activities. To validate our research hypothesis, we study two particular manifestations of developer communication, for which we are able to extract and synthesize data from the development repositories that report the day-to-day software development activities.

First, we develop tools and techniques to extract communication data from development repositories, and empirically validate these tools and techniques through case studies on development repositories of several open-source software projects. Next, we perform a linking of the extracted communication data back to the parts of the software system referenced within the data. These links enable us to investigate the impact of communication between stakeholders on software quality.

Second, we conjecture that a software system evolves through daily development activities that are captured on a technical level as changes (called patches) to the existing source code. Communication between developers materializes in the way through which this daily development effort is managed. To validate our research hypothesis, we propose to extract information about the management of code contributions, such as bug fixes, patches, or new features. We then derive a formal model of contribution management from a broad spectrum of open-source software systems. Based on this model, we perform an in-depth case study of two successful, large-scale open-source software systems. By contrasting management practices and processes, we are able to demonstrate that the management of contributions has a direct impact on the future inflow and integration of development effort into the software, and thus directly impacts the evolution of a software system.

1.2 *Thesis Organization*

In the following, we present an overview of the organization of this thesis. In particular, this thesis consists of five main chapters. Each chapter is dedicated towards a specific research problem. Our goal was to compose each chapter as a self-contained unit, such that readers can peruse each chapter independently. Thus, some repetition may exist between the various chapters, despite our best efforts to keep such repetition to a minimum. Furthermore, each chapter contains a separate discussion of related research work, tailored to the specific problem that is discussed in the corresponding chapter. We have organized the chapters of this thesis into four separate parts.

Part I of this thesis introduces the research problem, as well as our research hypotheses (*Chapter 1*). In addition, we present a broad background on the research topic, as well as a literature review and in-depth discussion of work related to this thesis in *Chapter 2*.

Part II of this thesis presents tools and techniques for mining collaboration data from software development repositories. In particular, we present a detailed discussion of the perils and pitfalls of mining communication data from e-mail communication archives (*Chapter 3*), as well as tools and techniques for separating natural language text from technical information (e.g., class names) contained in communication data (*Chapter 4*). In *Chapter 5*, we demonstrate through an empirical study that we can use the proposed tools and techniques to link communication data surrounding the software development process to the software product itself (i.e., the source code).

Part III of this thesis is dedicated to the empirical study of our hypothesis, using implementations of the tools and techniques form Part II. In *Chapter 6*, we use these links to explore the first part of our research hypothesis. In particular, we demonstrate that communication, among other social properties of development teams, has a strong relationship to software quality. software quality. In *Chapter 7*, we present an empirical study on how developer communication surrounding the management of code contributions impacts the evolution of a software system.

Part IV of this thesis provides an outline of future research directions in *Chapter 8*, together with a summary of the main contributions of this thesis, and our concluding remarks.

1.3 *Thesis Contributions*

This thesis makes a variety of contributions to the research field. In the following we highlight the key technical and conceptual contributions of each chapter.

1.3.1 *Technical Contributions*

1. *Chapter 3* demonstrates that communication data cannot readily be processed by traditional data mining, information retrieval and natural language processing approaches, and details common problems as well as possible technical solutions. Chapter 3 has led to the implementation of an email-mining tool named MailboxMiner[1], which is publicly available and has found extensive use in the research area, e.g., the work by German et al. and Jiang et al. [German2013a; Jiang2013].

2. *Chapter 4* contributes a lightweight approach to untangling unstructured data, in order to separate technical information from natural language text, as well as a manually developed benchmark suite to evaluate and compare the performance of future approaches against.

3. *Chapter 5* contributes an improved approach to link communication data to those parts of the source code that are being discussed. We demonstrate a sample application for visualizing which parts of the code are talked about the most.

4. *Chapter 6* contributes a novel set of socio-technical metrics surrounding the social interactions between developers. We demonstrate that these metrics can be used to understand software quality in a more holistic way when used in conjunction with traditional source code and process based metrics.

5. *Chapter 7* contributes a conceptual model of contribution management, as well as an investigation of successful practices of contribution management, which can be used by practitioners for establishing effective contribution management when moving towards an open-source business model.

[1] https://github.com/nicbet/MailboxMiner

1.3.2 *Conceptual Contributions*

1. *Chapter 3* demonstrates that communication data is unlike traditional software engineering data. Communication data exists as unstructured data that intertwines natural language text, project specific language, automated text, and technical information. We demonstrate that improper handling of this special kind of data can lead to substantial bias in data, experiments and results.

2. *Chapter 5* demonstrates that different conceptual classes of links between communication data and the software can be established. We further show that the approach presented in this chapter produces a novel class of traceability links that are fundamentally different from the links established by traditional approaches.

3. *Chapter 6* shows that three kinds of socio-technical relationships exist between collaboration and software quality. We demonstrate not only the existence of these relationships, but also their merits for creating defect models with explanatory power similar to traditional models based on product and process metrics. In addition, we show that a combination of these socio-technical metrics and traditional product and process metrics in defect models, yields higher explanatory power of software quality than taken separately.

4. *Chapter 7* demonstrates that timely communication is a key factor of successful contribution management. Our case study on two large open source software systems, makes a strong case for the importance of effective communication in order to prevent the waste of precious resources on dead-end code contributions.

Part I

Background and State-of-The-Art

2

Background and Literature Review

Past research has produced an extensive body of work describing relationships between socio-technical information about the software development process, the stakeholders surrounding software development activities, and the software product. This body of work consists of a multitude of unique data sources, data mining techniques, social and technical metrics, as well as a diverse set of modeling strategies, evaluations and recommendations to practitioners. In this chapter, we present a systematic survey of existing literature on the use of socio-technical information in software quality assurance, to compare and contrast prior work in this emerging research field, and to identify open research opportunities.

2.1 Introduction

Software development is a complex task. In order to break this complex task into smaller, manageable parts, software development efforts are often divided between individuals or groups of individuals. However, these individual development tasks do not exist in isolation: dependencies between development tasks drive a need for coordination and communication among software developers [Cataldo2006]. Organizational science has long recognized that social and communicational intricacies among team members impact task performance [Espinosa2002]. It is within this context that we arrived at our thesis hypothesis, that *"the communication between software developers plays a key role in the quality and the quality and evolution of a software."*. In this thesis, we observe developer collaboration through social, as well as technical information as recorded in development repositories. In the following, we give an overview of software quality assurance, as well as a short history of the emergence of socio-technical information in the software quality assurance process. The value of socio-technical information about software development efforts has been recently recognized by empirical software engineering research - in particular in the context of software quality assurance.

2.1.1 Contributions

The work presented in this chapter makes the following contributions to the research area: first, we present a systematic literature review on research that relates socio-technical information about software development efforts to software quality concerns. Second, we perform meta-analyses on the state-of-the-art literature and highlight the main findings, and identify future research opportunities in the research area.

2.1.2 *Organization of this Chapter*

This chapter is organized as follows. In Section 2, we present a brief background on the key concepts of literature covered by the literature presented in the rest of this chapter. In Section 3, we give a detailed description of the methodology that we followed to carry out our systematic literature review. In Section 4, we present a synthesis and analysis of the current research literature covered by our systematic review. We close this chapter with our concluding remarks, as well as highlighting the main findings of our literature review in Section 5.

2.2 *Background*

2.2.1 *Software Quality Assurance*

Software **Q**uality **A**ssurance (SQA), is an important means to ensure the quality of the final software product [Buckley1984]. In particular, SQA consists of techniques to audit and monitor the development process [Dawson2003], and methods to assess the product quality [Basili1996]. SQA ranges across the complete software development process [Ogasawara1996], from initial design, over requirements analysis, to the actual writing of source code, and subsequent testing, delivery and maintenance [Nagappan2006a].

In the past, researchers have proposed a large variety of software quality measures [Zimmermann2007; Basili1996; Dawson2003; Cataldo2009b; Cataldo2009a; Nagappan2007]. For example, a variety of studies investigate the relationships between characteristics of the software, such as its size [Fenton1991], complexity [McCabe1976], rate of changes [Nagappan2005], and post-release defects. Within our systematic review, we consider a broader definition of software quality, which encompasses both direct quality measures such as the number of defects in the final product delivered to the customer, as well as indirect measures such as design quality,

documentation ratio, or project health (see Table 2.10).

2.2.2 *Socio-Technical Information*

Software is the end-product of an involved, and highly collaborative effort of development teams [Ducheneaut2005; Bird2008; Cataldo2006]. For instance, Conway's law states that a software product reflects the organizational structure that produced it [Conway1968]. Around this observation, first published by Melvin Conway in 1968, a line of research named **"Socio Technical Congruence" (STC)** has emerged [Herbsleb2008]. In particular, research in the area of STC investigates, how closely technical dependencies (e.g., dependencies between source code files) mirror social dependencies (e.g., the organization of developers into teams responsible for sets of files) [Cataldo2008b], based on the proposition that good congruence between the two directly translates to an increased software quality [Kwan2011].

Research on socio-technical congruence has increased the awareness of the research community for the importance of social factors in software quality assurance. The past two decades in empirical software engineering have produced plethora of research work across a wide variety of socio-technical aspects for software engineering. In our systematic review, we consider a broader definition of Socio-Technical Congruence for this study than the original definition that is restricted to team organization. In particular, we consider Socio-Technical Congruence in this study as the degree to which the social information about a software, its development process, or the involved stakeholders influence and align with technical aspects of the software.

For example Cataldo et al. [Cataldo2009b] study the impact of work dependencies among developers on software defects, and de Souza et al. [Souza2004] study how information exchange among developers relates to problems in the development process. Grinter et al. [Grinter1999] study how communication issues among distributed teams lead to increased development costs. Kwan et al. [Kwan2011] investigate the impact of socio-technical issues on the build process, while Pattison et al. [Pattison2008] relate the communication about source code entities with

the software change process. In the same line of work, Bacchelli et al. [Bacchelli2010a] investigate whether source code entities that developers discuss have an increased risk of software defects.

2.2.3 *Mining Software Repositories*

Many studies on the use of socio-technical information for software quality assurance rely on the automated extraction of information from databases (i.e., repositories) that keep track of the software development process, also called software development repositories. To date, the research area of "**M**ining **S**oftware **R**epositories" (MSR) [Hassan2006], is concerned with tools and techniques to leverage historical development information from these repositories, in order to assist practitioners in future decision making [Xie2007].

The MSR area plays a key role in modern software quality assurance processes [Hassan2008; Hattori2008; Vandecruys2008; Aranda2009; Hassan2009; Zimmermann2007] and is the major source of technical information about the engineering process of a software product. However, social information is much harder to extract through mining software repositories, since much of the information is implicit, and often present in the form of unstructured (e.g., free-form natural language text) data [Bettenburg2010b]. Through our systematic review, we find that an overwhelming majority of research obtains socio-technical information through MSR methods, and that only a small subset of research employs data sources beyond software repositories, such as interviews or surveys (see Table 2.13).

2.2.4 *Definitions*

In the following we define and describe key concepts of our study.

> **Software Repository.** A software system or database that stores information surrounding the software development efforts of at least one developer.

Technical Information. Technical information surrounding software development, such as formal specifications, design, source-code, tests, examples, build-systems, or configuration files. In a broader sense, any technical aspect of the software development process. Table 2.7 presents a full overview of the technical information encountered by our systematic review.

Social Information. Any information surrounding the composition of the individual(s) carrying out the software development effort, such as team structure, personnel data, communication and coordination between individuals, discussions between stakeholders, email, and chat. Table 2.6 presents a full overview of the social information encountered by our systematic review.

Socio-Technical Information. Combinations of social information about a software, the development process, or the involved stakeholders and technical information on the software. To illustrate, consider the example of a study that would investigate how developer expertise relates to the likelihood of a developer producing a failure-free piece of software [Eyolfson2011]. Table 2.9 presents a full overview of the social-technical information encountered by our systematic review.

Software Quality. The conformance of a software to specified requirements, standards and implicit characteristics that are expected of professionally developed software [Pressman2001], i.e., the absence of software defects, as well as non-functional requirements, such as robustness, maintainability, security, efficiency, reliability and size. Within the context of our survey, we use a broader definition of software quality, such that software quality encompasses both external quality characteristics exposed to the user of the software, and internal quality characteristics [McConnell2004]. For instance, we considere development effort and implementation time as software quality measures that are not directly experienced by users, but both implicitly impact future changeability and maintainability of the software [Prause2012]. Table 2.10 presents a full overview of the software

quality metrics encountered by our systematic review.

Socio-Technical Network. Networks are commonly realized as a graph G with a set of edges E and a set of nodes N, and aim to capture relationships between observations about the software mathematically.

For instance, Zimmermann et al. [Zimmermann2008a] build a technical network that contains files as nodes, and edges between two files when there are data and module call dependencies. Zimmermann et al. subsequently use network metrics on this file-dependency network to predict the risk of a file containing a software defect.

On the other hand, Bird et al. [Bird2006b] mine a social network from email communication among developers, where nodes represent individual developers and an edge between two developers denotes that these developers communicated with each other. Bird et al. subsequently used network metrics to identify developers who are central to this communication network.

Socio-Technical networks are an extension to these two classical types of networks (technical network, social network), containing nodes that represent social, as well as technical information at the same time. For example, Pinzger et al. [Pinzger2008] built a socio-technical network with both developers, and files, as nodes, and edges between two file nodes, if there is a file dependency relationship, and edges between a developer node and a file node, if that developer changed that particular file. We want to note that in this particular socio-technical network, Pinzger et al. do not include edges between two developer nodes (which could for instance denote that developer A needed to coordinate with developer B). Pinzger et al. subsequently use network measures on this socio-technical network to build defect prediction models.

Social Network Analysis (SNA). Social Network Analysis is a set of mathematical techniques to study properties of social networks. Commonly, properties are defined as metrics

Metric	Definition	Example Interpretations
Degree Centrality	Number of incoming and outgoing edges of a node	Risk, Influence of a file, Popularity of a developer.
Closeness	The inverse of the distance between all-pair shortest paths	Ease of information spreading among developers.
Betweenness	The number of times a node acts as a bridge along the shortest paths of any two nodes	Importance of a developer for connecting development teams.
Network Centrality	Centrality of most central node in network in relation to centrality of all other nodes	Core-Periphery Structure of Development teams.
Hub / Bridge	Nodes that fill a structural hole	Team leads connecting two separate development teams.
Density	The proportion to which the number of edges is close to the maximum number of possible edges	How well connected development teams are.

Table 2.1: Summary of Prominent Social Network Analysis (SNA) Metrics Encountered in our Systematic Review.

that capture relationships within the social network graph, for instance the *centrality* metric is used to describe the importance of a node within the network, and is commonly measured as the number of incoming edges to that node, also called the "degree centrality" [Pinzger2008]. Articles covered in this systematic review describe a broad range of different SNA metrics, which in turn are connected to a variety of diverging interpretations as to what conceptual properties each metric aims to capture within the context of the particular social or socio-technical network presented in that article. We feel that listing all combinations of SNA metric definitions and interpretations of these metrics in software quality assurance literature is beyond the scope of this literature review, but poses a great opportunity for future research. However, we summarize in Table 2.1 the most prominent SNA metrics that we encountered during our systematic review.

2.2.5 *The Relationship Between Socio-Technical Information and Software Quality*

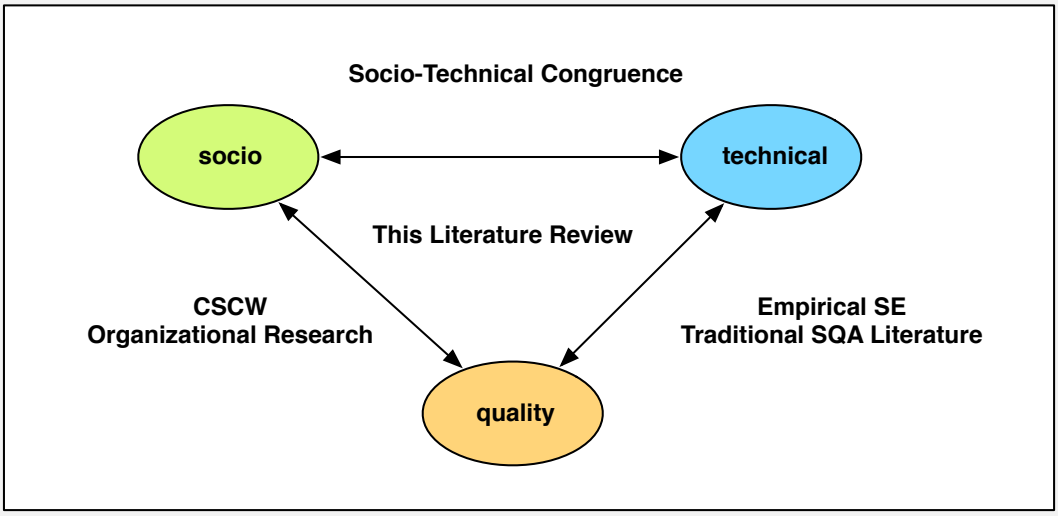


Figure 2.1: Graphical illustration of relationships between technical information, social information and software quality.

Past research in the area of software quality assurance can be categorized along three main research directions, as illustrated in Figure 2.1. On the one side, research investigating the relationships between software quality and purely social aspects of the software development process. For instance, the work by Di Penta et al. on the influence of communication overhead on staffing of software maintenance projects [DiPenta] falls in this category.

On the other side, work that is concerned with relating purely technical information about the software development process and the software product itself to software quality. For instance, the work by Mockus et al. [Mockus] describes how technical factors such as deployment time, operating system, or service contracts impact the perception of software quality from a customer perspective.

Third, work that is concerned with studying the relationships between social and technical aspects of software engineering, but without relating these aspects to software quality. For instance, Cheluvaraju et al. [Cheluvaraju] apply techniques from social network analysis on source code repositories to identify change dependencies and change propagation among source

code files, and Giger et al. [Giger] apply techniques from social network analysis on source code dependencies to predict software maintenance activities, such as declaration changes.

The literature review presented in this chapter focuses on the intersection of all three parts, i.e., the use of both, social and technical information for software quality assurance. In particular, our aim is to build and present a holistic overview on the use of socio-technical information in software quality assurance, by systematically selecting and reviewing research that considers combinations of both worlds, social and technical. For instance, the work by Herbsleb et al. [Herbsleb2003] investigates factors impacting development delays through examination of both, technical aspects of the development process, such as the size of changes or the number of source code files that were changed, as well as social aspects of the development, such as team size and geographical co-location of team members.

2.3 *Research Methodology*

The literature review presented in this chapter is carried out as a **S**ystematic **L**iterature **R**eview (SLR), which aims to identify, evaluate and compare available research related to the chosen research area in a documented, reliable and reproducible way. In particular our research methodology follows the guidelines on SLRs in computer science [Budgen2006; Kitchenham2009a], which have been demonstrated through a series of experiments to help capture the broadest review domain (maximizing recall), while optimizing review effort.

We adapted the guidelines described in previous research [Kitchenham2009a; Greenhalgh2005; Budgen2006; MacDonell2010] in a formal SLR process, which is illustrated in Figure 2.2. In the first step, we define a protocol for the SLR. In particular, this Section is a full written representation of that protocol.

Next, we define a set of research questions that on one hand guide our systematic review,

and on the other hand define the scope of our review. Based on this scope, we carry out an broad search of academic databases (INSPEC and COMPENDEX) for potentially relevant papers. The broad search is based on an iteratively refined search query, and results in a library of candidate papers for our SLR. We then proceed with a manual screening of all papers in the candidate library to identify and select relevant articles. From these relevant articles, we extract key topics and words, as well as references to articles that are cited by these relevant articles. With this information, we carry out a manual search through a broad range of conference proceedings to obtain an extended set of relevant articles.

In a third step, we perform a backward snowballing search through the extended library of relevant articles that were identified through the previous two steps, by examining all referenced research in these articles, as well as references of the referenced research. Through a manual screening process, we obtain a final set of primary studies that are included in our systematic review. With this final set of articles, we perform a systematic data extraction and mapping based on our six research questions. The result of this step is tabulated data, which in turn serves as a basis for our high-level data analysis, synthesis and presentation of our findings along the dimensions of the previously defined research questions.

2.3.1 Research Questions

To support the research objective of this systematic literature review, our process is guided by the following six research questions:

RQ1 What research exists in the field of empirical software engineering concerning the relationships between socio-technical information about the project and software quality.

RQ2 Which types of social and technical information about the software development process are covered by existing research?

RQ3 Which aspects of software quality are covered by existing research?

RQ4 What are the strategies, tools and data sources that are used to obtain socio-technical information about a software project?

RQ5 What methods are applied to study the relationships between social and technical information?

RQ6 What are the available future research opportunities ?

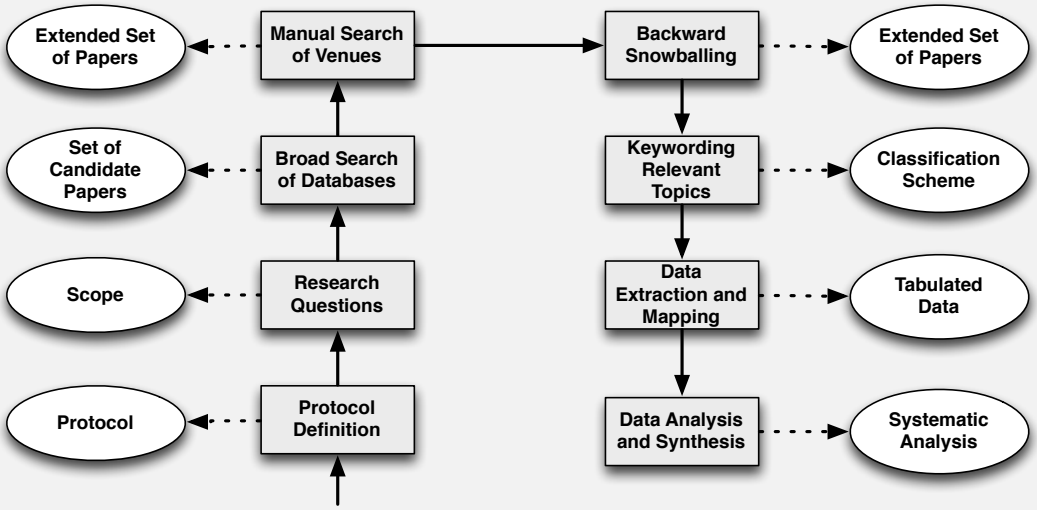


Figure 2.2: Systematic Literature Review - Process, adapted from [Kitchenham2009a]

2.3.2 *Broad Search of Academic Databases*

As a first step to identifying relevant research in the reviewed research area, we performed an broad search of two major academic databases. The **INSPEC** database is maintained by the Institution of Engineering and Technology[1] and contains a weekly updated index of over 11,000,000 articles from the multiple engineering-relevant disciplines, such as Physics, Electrical Engineering, Computing, Computer Science, Communications, and Information Technology.

The **COMPENDEX** database is maintained by Elsevier[2] and contains over 15,000,000 records from over 5,000 publication venues. The database specializes on engineering, computing, data processing, and computer science fields.

We accessed both databases through a common interface provided by the Engineering Village[3] web UI, accessed through the Queen's University Library Proxy. Engineering Village was bought by Elsevier in mid-2013 and is now a payed-access portal to the INSPEC and COMPENDEX databases.

[1]The Institution of Engineering and Technology - http://www.theiet.org
[2]Elsevier - http://www.elsevier.com
[3]The Engineering Village - http://www.engineeringvillage2.com

After multiple iterations of refinement, we arrived at the final search string presented in Figure 2.3 to query the academic databases for articles published between January 1st, 2000 and December 1st, 2012. In particular, our iterative refinement of the search string was performed such that we could ensure that the automated search would return five previously known key works of research in the designated review area [Bacchelli2010a; Bettenburg2010a; Cataldo2008a; Wolf2009; Mockus2000c]. In the following, we present and describe the final search string in detail.

```
(
  ({socio?technical} wn ALL)
  OR (social wn ALL)
  OR (social?network?analysis wn ALL)
  OR (SNA wn ALL)
) AND (
  (software?development wn ALL)
  OR (software?engineering wn ALL)
  OR (development wn ALL)
  OR (developer wn ALL)
  OR (developers wn ALL)
) AND (
  (repository wn ALL)
  OR (repositories wn ALL)
  OR (bugzilla wn ALL)
  OR (email wn ALL)
  OR (mailing?list wn ALL)
  OR (mailing?lists wn ALL)
)
```

Figure 2.3: Final Search String Used to Query the Academic Databases

The search string contains three main parts, separated by `AND`-clauses. The first part states that the article must include either term `socio-technical`, `social` or `social network analysis` (or the abbreviation SNA). We extended this part over multiple iterations, as the key phrase of "socio-technical" was coined in late 2007.

The second part states that the article must reference software development, or any popular synonym that expresses the same concept.

The third part states that the article must refer to a source of technical information, which

is commonly identified as a software repository. We have extended this part to explicitly include email, and BugZilla, a popular open source bug-repository, as these two sources were not formally considered software development repositories until the late 2000s.

We have modified the search string to allow for differences in spelling, expressed through the ?-sign separating individual key words. The term `wn` means that the phrase to the left of the term should be found in the entity to the right of the term, i.e., in our case we did not limit search to particular fields, but allowed matches in `ALL` records of meta-data attached to articles, such as Abstract, Title, Keywords, or Classification. Strings contained in {}-parentheses are searched as compound phrases (all individual sub-strings concatenated through AND), and all searches are case-insensitive.

The broad search of academic databases returned a set of 554 articles that matched the specified search string. After an additional manual pre-processing step of cleaning duplicate records (e.g., same article found in both databases, but spelling in title was slightly off - often in the case of non-alphabet characters), and non-articles (e.g., single abstracts submitted as workshop proposals, grant applications, etc.), we were left with a total of 410 articles that formed the set of candidate articles for manual screening.

2.3.3 *Manual Screening and Selection of Relevant Articles*

A major disadvantage of broach search of academic databases in systematic literature reviews is the low precision, i.e., many irrelevant articles are returned as the search string needs to be broad enough to obtain good recall [Jalali2012]. To prune the set of candidate articles and select those relevant to our study objective and research questions, we manually screened and filtered articles according to the following inclusion and exclusion criteria:

- **Inclusion Criteria.** The article should report on theory, practice, approaches, issues or opportunities regarding at least one source of technical information about a software project, and at least one source of social information about a software project. In addition,

the article should report on relationships between these two sources of information, socio-technical and software quality. Furthermore, the article should be unique, i.e., when a study is published in more than one venue (for example, conference papers and journal extensions of that conference paper), we considered the most complete version for analysis. Furthermore, the article should be published between January 1st, 2000 and January 1st, 2013.

- **Exclusion Criteria.** Articles describing only a single source of information about a software project, either social or technical. Articles that are irrelevant with respect to our research questions. Articles that do not describe a relationship between social and technical aspects surrounding a software project. Articles that do not describe relationships between social-technical aspects and software quality. Articles that describe tutorials, workshops, proposals, grants, posters, workshop summaries, keynotes, or single abstracts. Further, articles that lack scientific analysis or are published in unknown sources.

Examples of articles that matched the search string but were classified as irrelevant based on our inclusion and exclusion criteria were quite diverse and included themes such as:

- Nuclear Waste Disposal [JenkinsSmith2011]
- Building a Knowledge Management Library in Africa [Onyancha2012]
- Web 2.0 Ontologies [Boulos0703]

Table 2.2 presents an overview of the amount of articles filtered and selected through automated search, filtering based on meta-data such as keywords, abstract and title, and subsequent filtering based Overall, we removed a total of 287 irrelevant articles. After this initial culling step, we downloaded and read the remaining 123 articles in full text.

Of these 123 articles, 43 turned out to be unrelated to our research topic, 5 articles were New Ideas or Position Papers, and 29 articles considered only a single type of information (19

Step	n
Broad Search of Academic Databases	554
Removal of Duplicates	410
Selection of Articles based on Screening of Title and Abstracts	123
Selection of Articles based on Screening of Full Text	46
Included Articles	16

Table 2.2: Summary of Article Selection through Broad Search of Academic Databases and Subsequent Filtering According to Inclusion and Exclusion Criteria..

social, 10 technical) in software engineering. This leaves us a set of 46 articles from automated search that describe socio-technical information in software engineering. Of these 46 articles, 16 articles pass our inclusion criterium of describing relationships between the socio-technical information and software quality.

2.3.4 *Additional Search for Relevant Articles*

Kitchenham et al. demonstrated in 2009 that a broad search through academic databases is well suited for casting a broad net to quickly find relevant research in a research area [Kitchenham2009b]. However, the authors also demonstrated that a broad search through academic databases alone is at risk for claiming completeness when doing SLRs [Kitchenham2009b]. In particular, a manual search of conference and journal proceedings is often a necessary step when the sought-after research articles span multiple research areas, are part of an emerging research area (large collections often lag behind in what's available by up to a year), or have no well-defined established jargon that can be used for a reliable definition of a search string [Kitchenham2009b].

Additionally, MacDonell et al. showed that missing primary studies, which would have been found by including references of articles included from the automated search step (so-called "snowballing"), and manually searching for relevant articles through the repositories of major conferences and journals (so-called "broad manual search") have a significant impact on the

Abbrev.	Venue Full Name	*n*
APSEC	Asia-Pacific Software Engineering Conference	2
CIS	Conference on Information Systems	3
CSCW	Conference on Computer Supported Cooperative Work	2
CTS	Conference on Collaboration Technologies and Systems	2
GROUP	ACM Conference on Supporting Group Work	3
ICPC	International Conference on Program Comprehension	2
ICSE	International Conference on Software Engineering	6
ICSM	International Conference on Software Engineering	2
MSR	Working Conference on Mining Software Repositories	3
WCRE	Working Conference on Reverse Engineering	2
$\sum$		27

Table 2.3: Venues selected for Broad Manual Search, where *n* is the number of articles in that particular venue as identified as studying socio-technical information in the screening step of the broad search of academic databases (subsection 2.3.3).

results of SLRs [MacDonell2010]. Furthermore, Jalali et al. demonstrated that a composite search strategy that uses both a broad search through academic databases as well as snowballing yielded a higher recall of relevant articles than other search strategies [Jalali2012].

In the following we describe our search processes for both, our broad manual search through the repositories of major publishing vehicles, and a backward-snowballing search through all included relevant articles.

Broad Manual Search of Venues

We begin our broad manual search by determining a set of prospective venues that encourage articles describing socio-technical information. For each of the 46 articles identified to describe socio-technical information in the previous step, we recorded the venue that the particular article was published in. We considered a venue as relevant for inclusion in the broad manual search if two or more articles were published under that venue. Overall, we identified ten prospective venues that included 58% (27 out of 46) of articles concerning socio-technical information. Table 2.3 presents an overview of the venues selected for our broad manual search.

Following SLR guidelines that have been demonstrated as effective in previous research [Jalali2012], we downloaded records of Title, Abstract, Authors, and Keywords for all articles published under each venue between 2000 and 2013. We then performed a manual screening of all articles in accordance to our previously defined inclusion and exclusion criteria to select candidate articles for which to obtain the full-text. Based on thorough reading of the full-text we decided on relevancy of each article according to our inclusion and exclusion criteria. In particular, we defined the protocol for our manual search of venues through the following 5 steps. We repeat all five steps for each venue.

1. Define Search String specific to venue.
2. Perform search in COMPENDEX and INSPEC databases.
3. Screen Title and Abstract of results for relevant articles based on previously defined inclusion and exclusion criteria.
4. Download full text of the selected candidate articles.
5. Read full-text and decide on relevancy according to our inclusion and exclusion criteria.

Table 2.4 presents an overview of our selection of relevant articles by manual screening of articles published in the chosen venues. For example, the search through all conference proceedings of the International Conference on Software Engineering and co-located workshops (ICSE) was performed as a search through the INSPEC and COMPENDEX databases using the search string `((({International Conference on Software Engineering}) WN CF) AND ((ICSE) WN CF))`, which returned a total of 3,997 articles. Of these, we selected 69 candidates based on title and abstract and obtained the full-text for these articles. After screening the full-text, we selected 16 articles that described socio-technical information and had not been previously found by the automated search. Of these 16 articles, 9 articles related the socio-technical information to software quality and had not previously been found by our automated search.

Venue	Search Results	Candidates	Socio-Technical	Included
APSEC	1,711	18	2	0
CIS	2,021	12	0	0
CSCW	5,163	22	0	0
CTS	587	35	0	0
GROUP	692	14	0	0
ICPC	487	4	0	0
ICSE	3,997	69	16	5
ICSM	1,197	11	4	0
MSR	429	31	7	4
WCRE	767	5	1	0
$\sum$	17,051	221	28	9

Table 2.4: Screening for relevant articles in manual search of venues. The Included cloumn specifies the number of articles selected as relevant after screening of full-text and filtering according to our inclusion and exclusion criteria. Articles already included from the broad search of academic databases are not counted in this table.

Overall, our manual search of venues added a total of 9 new articles to our set of relevant articles that describe relationships between socio-technical information and software quality. Together with the 16 studies identified through automated search this brings our total to 25 relevant articles.

Backward Snowballing

In addition to performing a manual search through venues, we performed a manual screening of all articles that are referenced by the 74 articles (46 from automated search, 28 from manual search of venues) identified as describing socio-technical information from the previous two search steps (subsection 2.3.3 and subsection 2.3.4). This type of manual search is commonly referred to as "snowballing" [Kitchenham2009a]. In particular, we performed backward snowballing, i.e., starting from references in previously identified articles (as opposed to forward snowballing, which starts with citations to previously identified papers) [Greenhalgh2005], and we kept the snowballing depth to two levels, i.e., we considered direct references of included articles, as well as references of referenced articles.

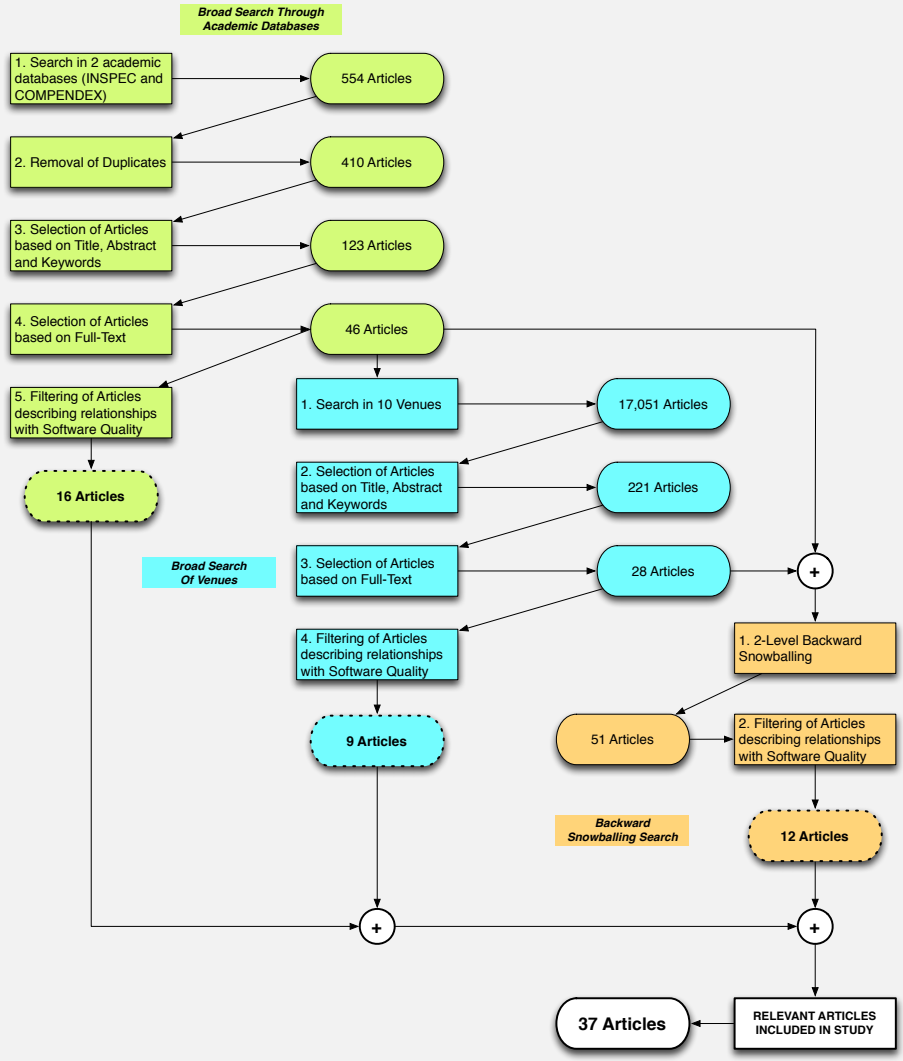


Figure 2.4: Summary of our Systematic Literature Search through the Three Phases of our Search for Relevant Articles.

Overall, we discovered 51 articles through snowballing that are concerned with socio-technical information. Of these, 19 articles describe relationships between socio-technical information and software quality. Of these 19 articles, 6 were already included in our set of relevant literature from the automated search and the manual search of venues step. Thus, snowballing added a total of 12 relevant articles. Together with 25 relevant articles identified

through automated and manual search, this brings our total to 37 relevant studies.

2.3.5 *Keywording and Card Sort*

This step of our systematic literature review protocol describes the process that we followed to systematically synthesize and extract data from the selected articles. In particular, we followed an Open Card Sort approach [Nawaz2012] to systematically distill appropriate dimensions for the dissemination of the research presented in the included articles.

We first performed a full-text reading of all selected articles, and recorded all descriptions of data sources, data gathering approaches, methodology, and other key research contributions that were related to our research questions on index cards. We then grouped these index cards into logical groupings, and labeled these groupings, such that they form categories that relate to our research questions. Figure 2.5 presents a complete map of the derived dissemination categories.

2.4 *Data Synthesis and Analysis*

In this section we present the data that we synthesized through the dissemination matrix, our analysis, as well as a critical discussion of our findings. In particular, we present our findings along the lines of our six research questions.

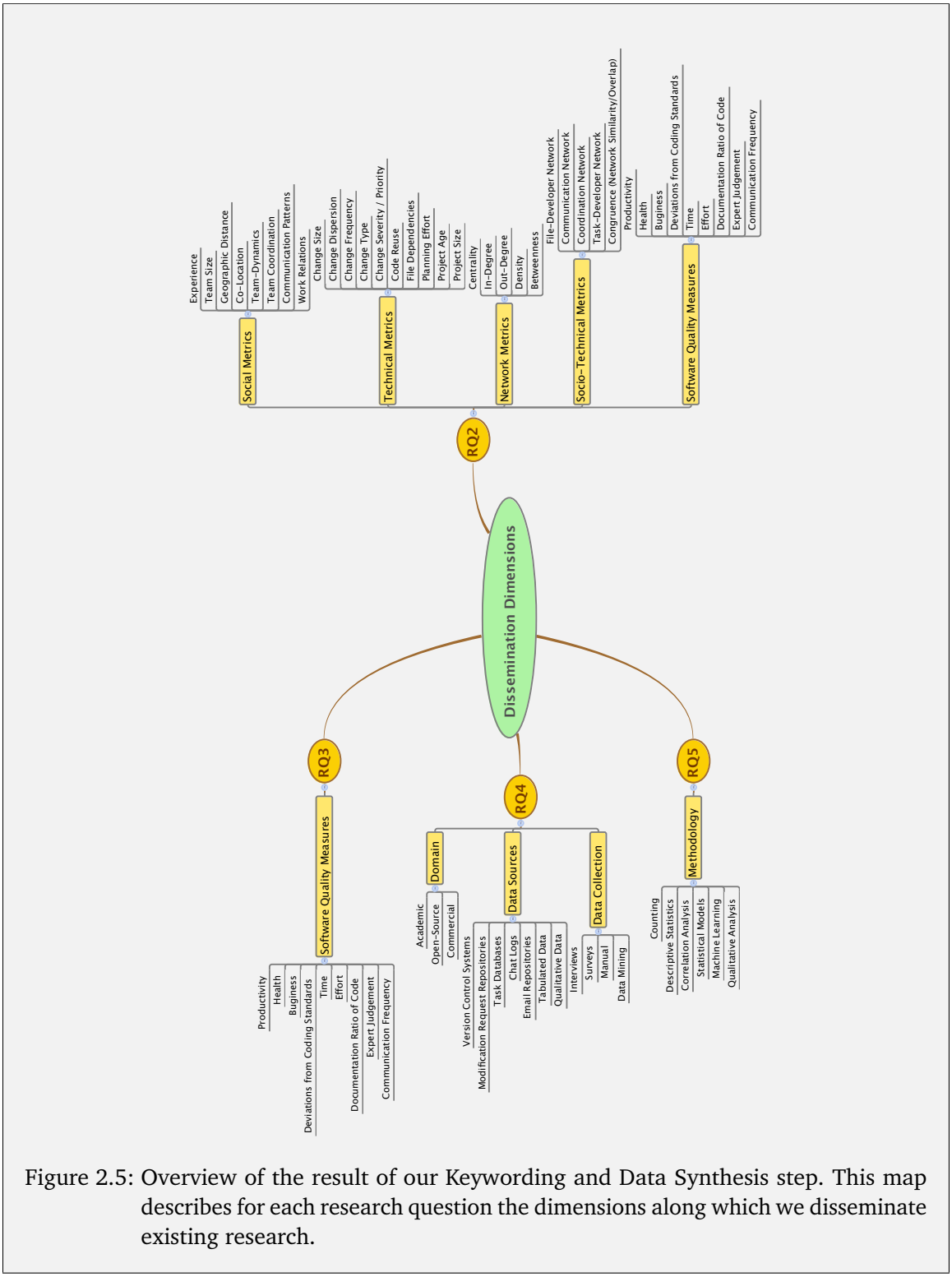


Figure 2.5: Overview of the result of our Keywording and Data Synthesis step. This map describes for each research question the dimensions along which we disseminate existing research.

2.4.1 *RQ1. What research exists in the field of empirical software engineering concerning the relationships between socio-technical information about the project and software quality.*

In the following we present summaries of all articles that were discovered through our systematic literature review. Each summary is intended to highlight data, methods and main findings of the corresponding article.

Summaries are grouped by their contributions to the advancement of what is known on socio-technical relations and software quality in the research area. Within each group, articles are ordered according to publication year, as we feel this ordering is important to understand the development of the research field from the initial hypotheses about socio-technical relationships derived from cognitive theory to the advanced analyses that merge multiple layers of socio-technical information into a single framework.

In addition, we would like to point our readers to the articles by Cataldo et al. [Cataldo2008b] and Nagappan et al. [Nagappan2008] that include tremendously insightful summaries of the fundamental theories from cognitive science that fueled and greatly influenced research in this area.

Summative Overview

In the following we present a summative overview of the literature included in this literature review. In particular, we have grouped studies by their primary study objective, the main quality metric involved, and the study domain. Our summary is presented in Table 2.5.

We find a total of nine top-level categories of study concerns, with varying levels of empirical evidence within their respective domains. For instance, we find a substantial amount of studies that investigate the applicability and performance of socio-technical concerns for defect prediction models, both in the industrial domain and the open-source domain. Our summative overview thus presents potential research avenues for future work in the area.

Table 2.5: Summative overview of Articles, by study concern and study domain.

Study Concern	Industrial	OSS
Socio-Technical vs. Development Times	[Espinosa2007; Herbsleb2003]	
Socio-Technical vs. Productivity	[Cataldo2008b; Ramasubbu2011; Nguyen2008; Ramasubbu2007; Cataldo2008a]	[Nguyen2008; Spinellis2006; Bird2006a]
Socio-Technical vs. Defect Density	[Cataldo2008b; Ramasubbu2011; Bird2009a; Cataldo2009a; Hulkko2005; Nagappan2008; Cataldo2006]	[Bird2012; Bettenburg2010a; Eyolfson2011; Hossain2009]
Socio-Technical for Defect Prediction	[Bicer2011; Bird2009c; Herbsleb2006; Meneely2008; Mockus2000c; Pinzger2008; Weyuker2007]	[Bicer2011; Bird2009a]
Socio-Technical for Vulnerability Prediction		[Meneely2009; Meneely2010; Shin2011]
Socio-Technical vs. Maintenance Activities		[Canfora2011]
Socio-Technical vs. Build and Integration Failures	[Cataldo2011; Kwan2011; Wolf2009]	[Wolf2009]
Socio-Technical vs. Design Quality		[Barbagallo2008; Barbagallo2009; Terceiro2010]
Socio-Technical vs. Project Health		[Amrit2010]

Socio-technical Relationships to Software Defects

Mockus et al. [Mockus2000c] present the use of statistical regression models to predict the likelihood of software development tasks containing errors based on product (e.g., number of lines of code) and process (e.g., defects delivered to the customer) metrics, as well as social metrics (e.g., developer experience). Mockus et al. identify the large number of files modified by the development task and low developer expertise as the two major flags to be presented to management for allocation of quality assurance resources.

Herbsleb et al. [Herbsleb2006] present a study that models coordination among software development teams as a distributed constraint satisfaction problem. Based on this model, they set out to investigate six research hypotheses that relate constraint density in this model to software quality and development efficiency. With respect to software quality, they find that the size of the development team making changes to the source code concerned with implementing a change (measured in number of people) have a significant impact on the risk of defects - in particular they find that for each additional developer who makes changes to a file, that file is 15 times more likely to contain defects.

> *The studies by Mockus [Mockus2000c] and Herbsleb [Herbsleb2006] are among the first to present empirical evidence that social information about developers can be combined with traditional defect prediction models based on source code and development process metrics. These socio-technical models have higher prediction accuracy than traditional technical models, and support the hypothesis that socio-technical relationships in the software development process impact software quality.*

Weyuker et al. [Weyuker2007] present an empirical case study on adding social metrics about developers of a software system to an existing defect prediction model that is based on traditional technical information metrics. Weyuker find that socio-technical defect prediction models outperform the fault prediction models that were built on technical information alone. In particular, their findings show that social metrics were the strongest explainers of software defects towards later releases.

Nagappan et al. [Nagappan2008] present an empirical study on the relationships between organizational properties of software development teams and failure proneness of the resulting software product. Through a case study on the Windows Vista system, they demonstrate that organizational properties were able to model failure proneness with higher precision and recall than state of the art prediction models based on process metrics (e.g., code churn) and product

metrics (e.g., CK complexity metrics).

Hossain et al. [Hossain2009] present an empirical study on the relationship between developer communication networks and software quality in Open Source Software projects. Through a case study on 45 open source projects, the authors demonstrate that social network analysis metrics measured from a task-developer network stand in a statistically significant relationship with code quality and defect density of open source software.

> *The studies by Weyuker [Weyuker2007], Hossain [Hossain2009] and Nagappan [Nagappan2008] present empirical evidence that socio-technical information can not only substitute, but also outperforms classical process and product based metrics when modeling software quality. The observations of all three studies make a strong case regarding the impact of socio-technical aspects in software development on the quality of the final software product.*

Bacchelli et al. [Bacchelli2010a] present an empirical study on the relationships between "popularity", measured through the frequency of communication between developers mentioning a specific technical entity like a class name, and the quality of that entity with respect to failure-proneness. Through a case study on 4 open source software systems, Bacchelli et al. find that entity popularity is strongly correlated to failure proneness. When used as additional dimensions in a traditional defect prediction model based on product and process metrics, Bacchelli et al. find that popularity metrics significantly increase the performance of these traditional models, while using popularity metrics alone results in performance comparable to traditional models. The results of their case study provides strong empirical evidence that failure-prone software entities are related to increased developer communication and collaboration efforts.

Bettenburg et al. [Bettenburg2010a] present an empirical study on the relationships between four dimensions of social metrics about the communication network and contents between

developers and software quality. Through a case study on the IBM Eclipse project, Bettenburg et al. find that socio-technical metrics explain software defects as good as defect prediction models based on traditional technical information alone. In addition Bettenburg et al. find that socio-technical metrics can significantly increase the performance of traditional models. Bettenburg et al. find that a measure of consistency of information flow in the communication between developers is among the strongest explainers of software defects. Their results provide empirical evidence of the impact of socio-technical relationships during software development on software quality.

> *While Bacchelli [Bacchelli2010a] demonstrates that an increased communication frequency about specific source code entities correlates with an increased defect density in these entities, Bettenburg [Bettenburg2010a] demonstrates that consistency in the communication flow is one of the strongest impactors of failure-proneness. Both studies support the hypothesis that developer communication flows and communication stand in relationship to software quality.*

Eyolfson et al. [Eyolfson2011] present an empirical study on the impact of developer experience and social environment on software quality. Through a multiple case study on two open source projects, the Linux kernel and the PostgreSQL database, Eyolfson et al. find that source code changes committed during the night hours (midnight to 4am) are significantly more likely to be defect prone than changes committed during early work hours (7am to noon). Eyolfson et al. control for developer experience, commit frequency, and day of week, and show that their findings hold against the control factors. Their study presents empirical evidence that environmental factors such as time of day have a measurable impact on software quality.

Co-Location of Developers

Herbsleb et al. [Herbsleb2003] perform a two-tier study on the effects of co-location of development teams on software quality. In the first part of the study, they model delays in the

development process by the means of a statistical model that includes social metrics (e.g., co-location, and size of development team) and technical metrics. In the second part of their study, they investigate the results of a survey that was performed at a development site where all developers were co-located and a remote development site. Their comparison centers around investigating the difference in social networks between co-located and remote sites. Their study identifies the lack of effective and frequent informal communication between developers in distributed teams as the main factor influencing development delays.

Hulkko et al. [Hulkko2005] present a multiple-case study on the impact of team-programming (i.e., Pair Programming) on software quality and development effort. They find that pair programming does not lead to code with less defects, but code with more code comments and also more deviations from coding standards. In addition to their quantitative study, Hulkko et al. conducted interviews concerning the rationale for pair programming. This qualitative part of the study revealed that pair programming requires most effort early in the development process, and is best suited for learning new complex code and programming tasks.

Spinellis et al. [Spinellis2006] present an empirical study on the effects of geographic distribution of the software development teams of the FreeBSD project on development productivity and product quality. They obtained file-developer and geographic networks through data mining of source code and task databases, and calculated geographic distances between pairs of developers in this network. In their three part case study, Spinellis et al. find no significant correlation between distance and defects, no correlation between distance and deviations from coding standards, and only a small correlation between distance and developer productivity. The authors note that their findings contradict earlier studies in the research area, and attribute these contradictions to the volunteer-driven, open-source development process of the FreeBSD project.

Espinosa et al. [Espinosa2007] present an empirical study on the relationships between socio-technical aspects of software development and the performance of software development teams. Within the context of their study, team performance is defined as the time it takes to implement all development tasks and remove software defects before release. Through a case study on the software development department of a commercial telecommunications company, they investigate how task and team familiarity, team size, geographic co-location and various technical process and product metrics impact team performance. They find that increased team familiarity is related to higher team performance, and geographic team dispersion is related to lower team performance.

Ramasubbu et al. [Ramasubbu2007] present an empirical study on the effect of co-location, team size, software reuse and different quality management approaches on both software quality, and development productivity. Within the context of this study, Ramasubbu et al. define conformance quality as the number of defects reported during acceptance tests and beta tests of the software product before final release. Through statistical modeling, the authors find that the dispersion of development across multiple sites has a negative impact on conformance quality and development productivity.

> *The work of Herbsleb [Herbsleb2003], Hulkko [Hulkko2005], Spinellis [Spinellis2006], Espinosa [Espinosa2007], and Ramasubbu [Ramasubbu2007] present empirical evidence that co-location of teams has a major impact on software quality. This effect is not so much rooted in the actual geographic location or distance, but in the complexity of organizational structures required for the management of the dispersed teams, and the resulting problems with team coordination and communication.*

Cataldo et al. [Cataldo2008a] present an empirical study on the impact of coordination in two

geographically distributed commercial software development projects on software quality and development productivity. They extract communication and coordination networks from IRC chat logs and Modification Request Tracking systems. Through regression models, Cataldo et al. demonstrate that two Social Network measures, centrality and network constraint, have a statistically significant impact on the development productivity of each project. The authors identify appropriate communication and coordination among development teams as essential for software quality and productivity.

Nguyen et al. [Nguyen2008] present a study on the impact of geographic distance on development productivity. In a case study on the IBM Jazz (now Rational Team Concert) product, Nguyen et al. extract the geographic location and communication networks of developers from the work item database that is integrated in the Jazz product. Contrasting same-site and distributed development the authors find that geographic distance does not have a detrimental effect on productivity. Nguyen et al. acknowledge that their findings are contrary to previous studies in the area and attribute this difference to the cognitive bias introduced in the survey-based data collection of those earlier studies on the one hand, and the unique development processes and tools of the IBM Jazz project on the other hand.

Bird et al. [Bird2009a] present an empirical study on the impact of geographical co-location of development teams on software quality. In a case study on the Windows Vista product, Bird et al. enhance a failure prediction model based on organizational metrics [Nagappan2008] with information about the geographic co-location of developers. Their findings demonstrate that co-located teams produced binaries with no statistically discernible difference in defect counts compared to teams that were distributed geographically. Bird et al. attribute their findings to the particular software development processes and practices at Microsoft that were designed to overcome communication and coordination barriers in distributed teams.

The studies by Bird [Bird2009a] and Cataldo [Cataldo2008a] present empirical evidence that coordination structure is more important than actual geographic distance in distributed software development. This evidence is hardened by the study by Nguyen et al. [Nguyen2008] who further show that tool support and, similarly to the findings of Bird et al. [Bird2009a], a modified software development process can diminish the impact of geographic distance on communication and development delays.

Cataldo et al. [Cataldo2009a] present an empirical study on the relationships between process maturity, geographic distribution of development teams, and software quality. In particular, Cataldo et al. focus on spatial distribution, as well as temporal distribution, in addition to the number of development locations and imbalances between these locations to model software quality. Through a case study on the software development department of a multinational automotive company. Through their findings, Cataldo et al. demonstrate a statistically significant impact of the number of development sites, as well as spatial and temporal dispersion of the development sites on software quality. They also find that uneven distributions of developers across development sites is connected to a decrease in software quality.

Cataldo et al. [Cataldo2011] present an empirical study on the relationships between socio-technical information and change integration failures in a feature-driven software development process. Through a case study on a commercial software project, Cataldo et al. find that organizational metrics about development teams contribute more explanatory power to the regression model than technical metrics. In addition, Cataldo et al. find that the geographic co-location of development teams is by far the highest impactor on change integration failures. Their findings present a strong case for the importance of developer coordination requirements in a feature-driven development process, i.e., geographically dispersed development teams with a low awareness of architectural coupling are detrimental to software quality.

The studies by Cataldo [Cataldo2009a; Cataldo2011] present multiple cases of empirical evidence that geographic dispersion of development teams in commercial software development projects have a strong impact on software quality, yet Bird [Bird2009a] and Nguyen [Nguyen2008] demonstrate that geographic distance has little impact on software quality in their respective case study subjects (both commercial). Bird argues in a later study [Bird2011] that as commercial software development processes become similar to open-source software development processes, issues caused by geographic and organizational dispersion diminish, but instead coordination becomes more challenging. Overall, work in this aread makes a strong case for organizational distance rather than geographic distance as the major impactor on software quality.

Ramasubbu et al. [Ramasubbu2011] present an empirical study on the impact of dispersion of development teams on software quality and productivity. Through a multi-case study on 362 software development projects from 4 different companies, Ramasubbu et al. derive different socio-technical configurations of the development teams. Their findings show that none of the derived configurations had beneficial outcomes across all three quality dimensions of productivity, software quality and profit-orientation, and as a result argue that globally distributed development always induces trade-offs between these three quality dimensions. In particular, Ramasubbu et al. note that while personnel imbalances increase productivity across the project, imbalances in team experience have a detrimental effect on productivity. At the same time these imbalances have the opposite effects on software quality, thus marking a significant conflict between productivity and quality.

Bird et al. [Bird2012] present an empirical study on the impact of geographical and organizational distribution in open source software projects on software quality. Through a case study

on IBM Eclipse and Mozilla Firefox, Bird et al. find that in Firefox, geographic distributions has only a small detrimental effect on software quality, and organizational distribution has no statistical effect, while in Eclipse geographic and organizational distribution have a detrimental effect on software quality. Bird et al. note that their study is the first to investigate geographical and organizational distribution within the context of open source development and software quality, and acknowledge a need for additional empirical results.

Socio-Technical Networks and Impact on Software Quality

Amrit et al. [Amrit2005] investigate the extent to which social network measures of centrality and density obtained from socio-technical networks connecting development tasks, developers and communication channels between developers are able to predict the performance of distributed software development teams. Performance is a composite measure consisting of the overall time taken to complete the development task, the quality of the delivered documentation, and the quality of the source code. They find that the proposed social network measures are predictors for development team performance and quality of the end product.

Barbagallo et al. [Barbagallo2008] present an empirical study on using social network analysis measures about the socio-technical network of open-source development teams to predict the design quality of software products. Through a case study on two samples of open source projects on SourceForge.net, they demonstrate that the centrality measure of the socio-technical network stands in significant correlation with project success, as well as project popularity with respect to the project's ability to attract open source contributors. They also find that the centrality measure of the socio-technical network has a negative effect on software design quality. Barbagallo et al. describe the centrality measure of the socio-technical network as an important variable to be monitored by project managers and team leaders.

The studies by Amrit [Amrit2005] and Barbagallo [Barbagallo2008] present strong empirical evidence that social network analysis is applicable to socio-technical relationships during software development. Overall, research has demonstrated that centrality, which describes nodes in the social network that are essential for coordination, is a valuable predictor for software quality. These findings confirm the hypotheses of the impact of coordination in software development on software quality, which have been postulated by previous research based on organizational science [Parnas1972; Kraut1995; Espinosa2002].

Bird et al. [Bird2006a] present an empirical study on the relationship between social status and development activity of developers in the Apache HTTP project. They mine data from version control archives and developer mailing lists and create a social network. They find that based on analysis of three social network metrics developers have a higher social status than non-developers, and that communication activity is highly correlated with development activity. In additions, Bird et al. find that development activity is a strong indicator of the social status of a developer in the project community. Both findings validate the previous hypotheses from cognitive science on the connection between developer communication and the product under development.

Meneely et al. [Meneely2008] present a study on using human factors in software development to predict software defects. For this purpose the authors propose an approach based on social network analysis of a developer-file network modelled from the changes recorded in a version control repository. Through a case study on a software product developed at Nortel, Meneely et al. find that the presence of developers who act as hubs in the developer-file network have a statistically significant impact on the defect density of files. Their case study demonstrates that social network metrics can be used instead of classical process metrics such as code churn for defect prediction and thus quality assurance efforts can commence early in the project, when

accurate code churn information is often not available to managers.

Pinzger et al. [Pinzger2008] present an empirical study on the relationship between social network analysis measures obtained from file-developer networks and failure-proneness of files. In a case study on Windows Vista, Pinzger et al. model file-developer relationships through contribution networks and demonstrate that centrality measures over the contribution networks have similar performance as organizational metrics [Nagappan2008], and outperform traditional process and product metrics with respect to failure prediction. The authors also demonstrate that social network metrics carry distinct information from classical product and process metrics, and can thus be used in addition to these classical metrics to enhance the performance of traditional prediction models.

The studies by Bird [Bird2006a] and Meneely [Meneely2008] show that social relationships between developers follow the small-world theory proposed by social science, and that some observations on organization and coordination theory are applicable to software development. In particular the study by Meneely [Meneely2008] demonstrates that social network metrics over these socio-technical networks can be used for defect prediction when classical product and process metrics are not available. Pinzger [Pinzger2008] further strengthens this empirical result and show that socio-technical information carries distinct information compared to information derived from classical product and process metrics.

Barbagallo et al. [Barbagallo2009] present an empirical study on the impact of developers who are hubs in the developer social network on software quality. Through a case study on 56 open source projects, Barbagallo et al. find that centrality is strongly related with developer experience, but at the same time centrality is associated with lower software design quality. The authors attribute the result to the coordination overhead caused by the central role that these

hub developers play in several development areas at the same time, and thus the overall lower time devoted to each individual area.

Bird et al. [Bird2009c] present an empirical study on the relationships between a combined socio-technical software network and fault-proneness of a software system. Through case studies on Windows Vista and IBM Eclipse, Bird et al. find that topological metrics derived from a socio-technical network that models file-dependencies and code contributions from developers consistently outperform traditional product and process metrics-based defect prediction models.

Meneely et al. [Meneely2009] present an empirical study on the impact of developer collaboration on software security. Through a case study on security issues of the Linux Kernel, Meneely et al. demonstrate that social network analysis metrics over file-developer networks have a strong statistical relationship with the occurrence of security issues in source code files. The case study explains the high vulnerability of files through unfocused contributions, i.e., changes carried out by developers who also changed many other files at the same time. Through their results, Meneely et al. see Brooks' law confirmed that simultaneous coordination effort developers need to spend across multiple tasks increases quadratically with the number of tasks, and as a result Meneely et al. argue that unfocused contributions pose a strong opposing force to Linus' Law of "many eyes make all bugs shallow."

The studies by Barbagallo et al. [Barbagallo2009], Bird et al. [Bird2009c] and Meneely et al. [Meneely2009] provide empirical evidence that developers who are central points, or "hubs" have a strong impact on software quality. The findings of these three studies show that even though such hub-developers play central supporting and essential roles, often bridging development efforts and teams, their division of attention across multiple coordination tasks creates choking points and the resulting unfocussed contributions negatively impact software quality in turn.

Wolf et al. [Wolf2009] present a study on using topology metrics over a developer communication and collaboration network to predict build failures in the IBM Jazz (now Rational Team Concert) project. Wolf et al. argue that their communication and collaboration network metrics capture coordination failures due to communication problems. Through their case study, Wolf et al. find that even though individual topology measures are unable to successfully predict build failures on their own, a combination of network measures is able to model build failures with precision and recall of up to 75%.

Amrit et al. [Amrit2010] present an empirical study on 8 open source software projects and investigate the relationships between core-periphery shifts in the socio-technical networks and the health of these projects. Within the context of their study, the health of a project is defined as the existence of a well-defined separation of roles of stakeholders according to the Onion model. Amrit et al. define a metric, Average Core Periphery Distance (CDPM), which describes how far each developer is from being part of the core of the socio-technical network. This metric is then averaged across all stakeholders in the project, and plotted over the course of multiple software releases. Amrit et al. identify 3 patterns (linear movement, oscillation, and steadiness) from these plots and through a qualitative analysis they validate these patterns as indicators of project health - steady movement away from the core, as well as oscillation are found to be indicators of unhealthy projects.

The studies by Wolf et al. [Wolf2009] and Amrit et al. [Amrit2010] are very different from prior work on the relationships between socio-technical artefacts and software quality. Wolf et al. [Wolf2009] are the first to use socio-technical information to predict the outcome of the software build process itself based on the previous communication and coordination between the developers working on the software, and Amrit et al. [Amrit2010] are the first to present empirical knowledge that project health is observable from socio-technical network metrics.

Meneely et al. [Meneely2010] present a replication study on their earlier work ([Meneely2009]), extending their case study to two additional open source projects. Through the extended multiple-case study, Meneely et al. find that the findings of their previous work generalize to the additional case study subjects, and that models learned on one case study system can be transferred to another case study system with comparable performance. Their results thus provide empirical evidence to the generalizability of the impact of unfocussed contributions on software vulnerability.

Terceiro et al. [Terceiro2010] present an empirical study on the social structure of open source software development teams on software quality. In particular, Terceiro et al. investigate the differences between core and periphery developers as defined in the Onion model, with respect to the structural complexity that developers introduce when changing the software product. Through a case study on 7 open source web server projects written in the C language, Terceiro et al. find that changes carried out by core developers have less negative impact on the quality of the software than changes carried out by periphery developers. In addition, Terceiro et al. find that refactorings carried out by core developers result in higher software quality than those carried out by periphery developers.

Bicer et al. [Bicer2011] present a study of defect prediction using Social Network Analysis metrics over the Developer Communication networks mined from Modification Request repositories.

Through a multiple case study on Rational Team Concert, a commercial project, and Drupal, an open-source project, Bicer et al. demonstrate that a Naive Bayes predictor learned from SNA metrics significantly outperforms traditional predictors based on process metrics in both case study subjects. In addition, Bicer et al. perform a cost-benefit analysis and contrast the socio-technical prediction model with the traditional technical model and find that the socio-technical model provides an increased prediction performance at a lower cost (in terms of quality assurance effort). Bicer et al. argue that their results demonstrate that information flow in the developer communication network significantly impacts software quality.

Canfora et al. [Canfora2011] present a study on the social characteristics of developers who are involved in cross-system corrective maintenance. For this purpose, Canfora et al. construct a developer communication network for the developers in two open source projects, FreeBSD and OpenBSD, and relate social network metrics to cross-system bug fixing changes. The findings of their case study demonstrate that developers involved in cross-system bug fixing exhibit a larger communication network, are central for information flows and act as bridges between both projects. In addition, Canfora et al. find that developers involved in cross-system bug fixing activities are among the most productive developers with respect to the amount of modified source code lines. Their work identifies authors of cross-system corrective maintenance as key actors for both information flow, and brokerage of coordination.

Shin et al.[Shin2011] present an empirical study on the relationships of socio-technical aspects of open-source development on software vulnerabilities. In particular, Shin et al. combine CK metrics with SNA measures over file-developer networks and use a Bayesian classifier to predict vulnerabilities. Through a multiple case study on Mozilla Firefox and the RedHat Linux 4 Kernel. Shin et al. demonstrate that CK metrics alone fail to consistently predict vulnerabilities. Furthermore, the authors argue that vulnerability detection and defect detection might be distinct problems, and that technical metrics that are successful predictors in one domain might not generalize to the other domain.

Socio-Technical Congruence and Impact on Software Quality

Cataldo et al. [Cataldo2006] present an empirical study on the relationships between development task dependencies and developer coordination on software quality. They capture the similarity between both types of dependencies in congruence measures, which are used in a set of linear regression models to analyze their relationship to the time it took developers to complete bug fixing tasks. Cataldo et al. find a consistent strong statistical correlation between faster removal of bugs and high congruence, i.e., the better the coordination of development teams fits their coordination requirements, the faster they were able to remove bugs before software releases, thus increasing software quality.

Cataldo et al. [Cataldo2008b] build upon their earlier work ([Cataldo2006]) and define a new metric, socio-technical congruence that measures how well the technical and social properties of the software product under development align. The congruence measure proposed in their work is based on a coordination requirements matrix, which is built from multiple data sources, such as task assignment records, task dependency networks, communication archives, and organizational structure. This congruence measure is used together with control factors like task priority, change size, and programmer experience in a set of regression models to predict the time taken to implement changes in the software product. Cataldo et al. argue that alignment of socio-technical aspects is a key part of coordination in software development and demonstrate through a case study that their congruence metric stands in a strong statistical relationship with development productivity.

The two studies by Cataldo [Cataldo2006; Cataldo2008b] formalize Conway's law into a single metric of congruence that measures how well the social and technical aspects of a software development effort align. In particular, their 2008 study [Cataldo2008b] demonstrates that such a congruence metric does successfully capture misalignments in development coordination and stands in strong statistical relationship with development productivity.

Kwan et al. [Kwan2011] present a study on the relationships between socio-technical congruence measures and build failures. In particular, Kwan et al. augment the original congruence measure proposed by Cataldo et al. through the introduction of weights that allow the identification of relative mismatches between social and technical dimensions. Through a case study on a commercial software product, Rational Team Concert (RTC), Kwan et al. demonstrate that for continuous build types, increased congruence improves the chance of build success, but decreases the chance of build success in integration build types. Overall, Kwan et al. found that components in the RTC project have overall low congruence, which is considered as detrimental to software quality. However, Kwan et al. argue that the particular software process and tool support of the Rational Team Concert project aim to provide developers with strong awareness and enforces explicit communication, which allows developers to be aware of builds that require high coordination, and thus counters the detrimental effects of low overall congruence.

2.4.2 *RQ2. Which types of social and technical information are covered by existing research?*

We found a broad range of socio-technical information about software development covered in the surveyed literature. We classify socio-technical information into the following four categories: social information about the software project, technical information about the project, combinations of social and technical information into network structures, and the corresponding social network analysis metrics used to investigate those network structures. Within each category, we give a short description of the social or technical information. For each item, we also list the relative amount of studies (percentage, where 100% denotes a coverage by all 37 articles that were included in this systematic review) that included that particular information item.

Social Information

Table 2.6 summarizes the social information encountered in the articles covered by our systematic review. We provide a short description below each social information item.

Co-Location of Developers (27%) captures whether developers or development teams are located within a similar geographic entity. This information is used both as a binary variable, e.g., [Espinosa2007], as well as a factor variable denoting the level of co-location, such as "same office", "same floor", "same building", and "same city" [Bird2009c].

Number of Development Sites (8%) captures the number of development sites, that form a single entity of co-location, and that are involved in the development process. Cognitive theory describes that coordination within one site is easier than coordination across different sites [Cataldo2009a].

Communication Patterns (8%) capture specifics of the communication processes such as contents [Bacchelli2010a; Bettenburg2010a], or communication disruptions expressed through entropy [Bettenburg2010a].

Developer Experience (27%) captures the familiarity of developers with software components, tasks, or APIs [Mockus2000c].

Developer Skill Level or Skill Range (2%) captures the technical expertise of developers [Barbagallo2009].

Geographic Distance between Developers (16%) captures the distance "as the bird flies" between developers or development sites [Bird2009c].

Time of Day or Day of Week of Development (2%) denotes the time and weekday at which development effort was carried out [Eyolfson2011].

Developer Native Language and Culture (2%) captures differences in language and culture within development teams [Herbsleb2003].

Number of Developers (54%) denotes the amount of distinct engineers responsible for a particular software development effort [Mockus2000c].

Number of Developers who Left the Project (2%) denotes the amount of engineers who left the project and is intended to capture the loss of knowledge in the development project [Nagappan2008].

Team Coordination Metrics (8%) capture a variety of coordination metrics, and can range from qualitative data, e.g., on a Likert scale from "easy" to "hard" [Herbsleb2003], to distinctions of being part of the core or periphery of a development effort [Terceiro2010].

Organizational Metrics (16%) capture a variety of organizational metrics such as the proportion of the organization contributing to the development effort [Nagappan2008], or process maturity level within the organization [Cataldo2009a].

Work Relations (5%) describe how comfortable developers feel in their work environment [Herbsleb2003], or how often they seek advice from colleagues [Amrit2005].

Table 2.6: Dissemination Matrix: Social Information (RQ2)

Reference	Co-Location	Number of Development Sites	Communication Patterns	Developer Experience	Developer Skill Level / Skill Range	Geographic Distance	Time of Day / Day of Week	Language / Culture	Number of Developers	Number of Ex-Developers	Team Coordination Metrics	Organizational Metrics	Work Relations
[Mockus2000c]				✓					✓				
[Herbsleb2003]	✓		✓					✓	✓		✓		✓
[Amrit2005]	✓												✓
[Hulkko2005]											✓		
[Bird2006a]													
[Cataldo2006]				✓									
[Herbsleb2006]									✓				
[Spinellis2006]						✓							
[Espinosa2007]	✓								✓				
[Ramasubbu2007]	✓								✓				
[Weyuker2007]				✓					✓				
[Barbagallo2008]													
[Cataldo2008a]				✓									
[Cataldo2008b]	✓			✓									
[Meneely2008]									✓				
[Nagappan2008]				✓					✓	✓		✓	
[Nguyen2008]	✓	✓				✓			✓				
[Pinzger2008]									✓				
[Barbagallo2009]				✓	✓								
[Bird2009a]	✓					✓			✓			✓	
[Bird2009c]													
[Cataldo2009a]		✓				✓			✓			✓	
[Hossain2009]													
[Meneely2009]									✓				
[Wolf2009]									✓				
[Amrit2010]													
[Bacchelli2010a]			✓										
[Bettenburg2010a]			✓						✓				
[Meneely2010]									✓				
[Terceiro2010]											✓		
[Bicer2011]													
[Canfora2011]													
[Cataldo2011]	✓			✓					✓			✓	
[Eyolfson2011]				✓			✓						
[Kwan2011]									✓				
[Ramasubbu2011]	✓	✓		✓		✓			✓			✓	
[Shin2011]									✓				
[Bird2012]	✓					✓						✓	

Technical Information

Table 2.7 summarized the technical information encountered in the articles included in our systematic review. We provide a short description below each technical information item.

Change Dispersion (14%) captures how spread out development efforts are [Mockus2000c].

Change Frequency (43%) captures how frequently did software entities change within a set period of time. This metric is also known as "churn" metrics [Nagappan2008].

Change Size (35%) captures the size of a software development effort under study. Size is commonly expressed as the number of lines of code added, deleted, or modified (i.e., added+deleted) [Mockus2000c].

Change Type (16%) indicates the type of the software development effort under study. Examples are corrective changes ("bug fixes"), or implementation of new features [Mockus2000c].

CK Metrics (22%) describes any product metric as defined in the Chidamber and Kemerer metrics suite [Chidamber1994].

Code Reuse (2%) indicates whether code reuse was part of the software process or not [Ramasubbu2007].

Modification Request Priority or Severity (11%) indicates the importance of a modification request [Mockus2000c].

Number of Past Defects (10%) indicates the amount of defects recorded in the past for a particular software entity [Mockus2000c].

Planning Effort Spent (2%) captures how much effort was spent during the planning phase of the project, and acts as a proxy to the initial design quality [Ramasubbu2007].

Project Size (14%) captures the total size of the software project measured in lines of code [Ramasubbu2007].

Unit-Test Coverage of the Software (2%) captures the amount of source code covered by the unit-testing framework [Bird2009a].

Software Age (8%) captures the total age of the software. Aging software has more structural complexities, and might be harder to change and maintain [Espinosa2007].

Socio-Technical Networks

Table 2.9 presents a summary of the socio-technical networks and the corresponding network metrics encountered in the articles included in our systematic review. We provide a short description of each social-technical network. Furthermore, Table 2.8 presents summaries on the node and edge types of each socio-technical network to ease readability.

File-Dependency Network (19%) describes a network with files that are part of the software development effort as nodes. Edges denote relationships between these files. For example, direct file relationships such as imports in source code files [Herbsleb2006], or files that are connected because the same developer carried out changes [Meneely2010].

File-Developer Network (43%) describes a network with files, as well as developer identities as nodes. Edges denote any kind of relationship between nodes, such as direct file-to-file dependencies and file-change information [Cataldo2008a].

Table 2.7: Dissemination Matrix: Technical Information (RQ2)

Reference	Change Dispersion	Change Frequency	Change Size	Change Type	CK Metrics	Code Reuse	CR Severity/Priority	Number of Previous Defects	Planning Effort	Project Size	Unit-Test Coverage	System Age
[Mockus2000c]	✓		✓	✓								
[Herbsleb2003]	✓		✓	✓			✓					
[Amrit2005]												
[Hulkko2005]												
[Bird2006a]												
[Cataldo2006]			✓	✓			✓					
[Herbsleb2006]	✓	✓	✓	✓								
[Spinellis2006]												
[Espinosa2007]			✓	✓								✓
[Ramasubbu2007]						✓			✓	✓		
[Weyuker2007]		✓	✓					✓				
[Barbagallo2008]					✓					✓		✓
[Cataldo2008a]			✓									
[Cataldo2008b]			✓				✓					
[Meneely2008]		✓										
[Nagappan2008]		✓			✓			✓				
[Nguyen2008]							✓					
[Pinzger2008]		✓										
[Barbagallo2009]					✓							
[Bird2009a]		✓			✓						✓	
[Bird2009c]												
[Cataldo2009a]		✓	✓	✓						✓		
[Hossain2009]												
[Meneely2009]		✓										
[Wolf2009]		✓								✓		
[Amrit2010]												
[Bacchelli2010a]		✓	✓		✓			✓				✓
[Bettenburg2010a]								✓				
[Meneely2010]		✓										
[Terceiro2010]					✓							
[Bicer2011]		✓										
[Canfora2011]												
[Cataldo2011]	✓		✓		✓							
[Eyolfson2011]		✓										
[Kwan2011]	✓	✓										
[Ramasubbu2011]										✓		
[Shin2011]		✓	✓		✓							
[Bird2012]		✓	✓									

Type	Node	Edge
File-Dependency Network	File	Direct File Dependency Relationships.
File-Developer Network	File or Developer	Direct File Dependencies or Authorship.
Task-Developer Network	Tasks or Developers	Coordination Requirements.
Communication Network	Developers	Information Exchange between Developers.
Geographic Network	Locations or Developers	Weighted by Geographic Distance.
Organizational Network	Organizational Units or Developers	Contains Relationships.

Table 2.8: Summary of Socio-Technical Networks Encountered in This Systematic Review.

Task-Developer Network (30%) describes a network with development tasks, as well as developer identities as nodes. Edges between nodes denote any kind of relationship, such as coordination requirements [Amrit2005].

Communication Network (35%) describes a network with developer identities as nodes. Edges between nodes denote communication exchanges between developers [Amrit2005].

Geographic Network (14%) describes a network with geographic locations and developers as nodes. Edges between locations can be weighted by the geographic distance between these locations [Cataldo2006].

Organizational Network (11%) describes a network with organizational units and developer identities as nodes. Edges usually denote “contains” relationships [Barbagallo2008].

Table 2.9: Dissemination Matrix: Socio-Technical Networks and Measures (RQ2)

Reference	File-Dependency Network	File-Developer Network	Task-Developer Network	Communication Network	Geographic Network	Organizational Network	Congruence	SNA Betweenness	SNA Centrality	SNA Density	SNA Degree	Number of Network Hubs
[Mockus2000c]												
[Herbsleb2003]												
[Amrit2005]			✓	✓					✓	✓		
[Hulkko2005]												
[Bird2006a]				✓				✓			✓	
[Cataldo2006]			✓	✓	✓		✓					
[Herbsleb2006]	✓	✓									✓	
[Spinellis2006]		✓			✓							
[Espinosa2007]			✓	✓								
[Ramasubbu2007]												
[Weyuker2007]												
[Barbagallo2008]		✓				✓		✓	✓		✓	
[Cataldo2008a]			✓	✓	✓							
[Cataldo2008b]	✓	✓	✓	✓	✓	✓	✓					
[Meneely2008]		✓						✓			✓	✓
[Nagappan2008]						✓						
[Nguyen2008]				✓	✓	✓						
[Pinzger2008]		✓						✓	✓		✓	
[Barbagallo2009]			✓					✓	✓		✓	✓
[Bird2009a]	✓										✓	
[Bird2009c]	✓	✓						✓	✓			
[Cataldo2009a]	✓											
[Hossain2009]			✓					✓	✓	✓		
[Meneely2009]		✓						✓				
[Wolf2009]		✓	✓	✓				✓	✓		✓	
[Amrit2010]	✓	✓							✓			
[Bacchelli2010a]			✓	✓								
[Bettenburg2010a]			✓	✓					✓			
[Meneely2010]	✓	✓						✓				
[Terceiro2010]		✓										
[Bicer2011]		✓		✓				✓	✓	✓	✓	
[Canfora2011]		✓		✓				✓	✓	✓	✓	
[Cataldo2011]												
[Eyolfson2011]												
[Kwan2011]	✓	✓	✓	✓			✓					
[Ramasubbu2011]												
[Shin2011]	✓	✓						✓	✓		✓	
[Bird2012]												

Network Measures

Network Congruence (8%) measures the extent to which the social and technical information within socio-technical networks overlap [Cataldo2008a]. The framework of socio-technical congruence conjectures based on Conway's Law that deviations from ideal overlap are connected to coordination problems which in turn lead to detrimental effects on software quality [Kwan2011].

SNA Betweenness (35%) denotes the extent to which individuals in the social network act as brokers or gatekeepers between otherwise disconnected parts of the network [Bird2006a].

SNA Centrality (32%) denotes the extent of popularity of individuals in a social network [Bicer2011].

SNA Density (11%) denotes the extent of overall connections between individuals in the social network [Amrit2005].

SNA Degree (30%) denotes the extent to which individual nodes in the social network are connected to other nodes [Bird2006a].

SNA Number of Hubs (5%) measures the number of developers with higher than normal betweenness [Meneely2008].

As mentioned earlier, articles covered by our systematic review contain interpretation of social network analysis (SNA) measures that are specific to the context of each individual article. As Amrit et al. note, differences with respect to the meaning and interpretation of the studied social network metrics are not unexpected, as even the research field of social network theory contains much discourse on the various interpretations of social network measures [Amrit2005].

We observe a wide range of socio-technical information described in literature. However, many potentially useful metrics remain limited to single studies. For instance, social metrics such as knowledge loss through developers who have left the project, or developer skill level, are described in only a single study. We believe that a broad range of socio-technical metrics remains unexplored and that future research could benefit from additional empirical work. Furthermore, we observe a distinct lack of metrics describing the communication between developers, beyond basic SNA metrics. ***In Chapter 6 of this thesis, we introduce a novel set of socio-technical metrics describing developer communication, and investigate to what extent these metrics describe software quality.***

2.4.3 *RQ3. Which aspects of software quality are covered by existing research?*

Table 2.10 presents a summary of the quality metrics encountered in the articles included in our systematic review. Overall, we encountered a broad variety of software quality metrics discussed in literature. In the following we describe each metric in more detail.

Defect Density (49%) is the most popular software quality metric and is a direct measure of the quality of the software. The studies that we encountered provide empirical evidence that defect density in a software product is not only impacted by technical aspects of the development process, but it is also strongly impacted by a variety of social aspects such as developer experience [Mockus2000c], organizational concerns such as the organization hierarchy [Bird2009a] and code-ownership [Nagappan2008], the coordination among developers [Pinzger2008], communication among developers [Herbsleb2006; Bettenburg2010a], and even the time of the day that development is carried out [Eyolfson2011].

We have found empirical evidence that social information describes aspects of the software

Table 2.10: Dissemination Matrix: Software Quality Measures (RQ3)

Reference	Defect Density	Vunerability	Build / Integration Failures	Communication Frequency	Design Quality	Development Effort	Deviations from Coding Standards	Documentation Ratio	Expert Judgement	Productivity	Project Health	Profit	Time to Implement
[Mockus2000c]	✓												
[Herbsleb2003]				✓									✓
[Amrit2005]									✓				✓
[Hulkko2005]	✓					✓	✓	✓					
[Bird2006a]										✓			
[Cataldo2006]													✓
[Herbsleb2006]	✓					✓							✓
[Spinellis2006]	✓						✓			✓			
[Espinosa2007]										✓			✓
[Ramasubbu2007]										✓			
[Weyuker2007]	✓												
[Barbagallo2008]					✓	✓							
[Cataldo2008a]										✓			
[Cataldo2008b]										✓			
[Meneely2008]	✓												
[Nagappan2008]	✓												
[Nguyen2008]				✓									✓
[Pinzger2008]	✓												
[Barbagallo2009]					✓								
[Bird2009a]	✓												
[Bird2009c]	✓												
[Cataldo2009a]	✓												
[Hossain2009]	✓												
[Meneely2009]		✓											
[Wolf2009]			✓										
[Amrit2010]											✓		
[Bacchelli2010a]	✓												
[Bettenburg2010a]	✓												
[Meneely2010]		✓											
[Terceiro2010]					✓								
[Bicer2011]	✓												
[Canfora2011]										✓			
[Cataldo2011]			✓										
[Eyolfson2011]	✓												
[Kwan2011]			✓										
[Ramasubbu2011]	✓									✓		✓	
[Shin2011]		✓											
[Bird2012]	✓												

development process that are not captured by traditional process and product metrics, evidenced both through increased explanatory power [Bettenburg2010a] of defect prediction models, as well as significantly increased prediction performance [Bicer2011]. The studies captured in this literature review make a strong case for the role of socio-technical information in both, understanding the nature of software defects, and software quality assurance through defect prediction.

Vulnerability (8%) describes how likely a software product is to concern security-critical issues. Shin et al. [Shin2011] argue that vulnerability captures a conceptually different aspect of software quality than defect density. Shin et al. demonstrate through their case study that traditional product metrics cannot produce meaningful models for predicting vulnerabilities even though such metrics perform well in predicting software defects. Both Meneely et al. [Meneely2010] and Shin et al. [Shin2011] demonstrate that a combination of social and technical information can be used to predict vulnerabilities with moderately high performance. In particular, Meneely et al. [Meneely2009] isolate unfocussed contributions, i.e. contributions by developers who have to share their attention across an above-average amount of coordination tasks, as a key factor for the occurrence of software vulnerabilities. Thus their findings provide another point of validation for the hypothesis that coordination plays a major role in connection with software quality.

While **Build and Integration Failures (8%)** do not describe software quality aspects that directly concern the end-user, they are still of high interest to the producers of a software product [Wolf2009]. However, build failures indirectly impact the software quality tangible by end-users as build and integration failures impact development productivity and timelines, and consume effort that could otherwise be spend on software quality assurance [Kwan2011]. All three studies presented in this literature review [Wolf2009; Cataldo2011; Shin2011] demonstrate that socio-technical information can better explain and predict build and integration failures

compared to classical technical information-based models. The findings of these studies provide another point of empirical evidence that social information about software development aspects plays a major role in modeling the outcomes of software development efforts.

Communication Frequency (5%), while not a direct measure of software quality, is described as an indirect concern for software quality [Herbsleb2003; Nguyen2008]. In particular, increased communication frequency is show to stand in connection with increased coordination [Herbsleb2003], which in turn has been shown to be a direct factor that impacts software quality [Herbsleb2003; Cataldo2008b]. In addition, the study by Bettenburg et al. [Bettenburg2010a] demonstrates that communication frequency is one of the strongest predictors of defect density, in particular when communication frequency shows irregularities.

Design Quality (8%) is a direct measure of software quality, and directly impacts the maintainability of the software product on the one hand, and the risk of subsequent changes introducing software defects on the other hand [Barbagallo2009]. All three studies [Barbagallo2009; Barbagallo2008; Terceiro2010] captured in this literature review show that socio-technical concerns stand in direct relationship with design quality. However, the studies captured in this literature review did not agree on this relationship. For instance, [Barbagallo2008] and [Barbagallo2009] attribute modifications to the software carried out by developers that have a central place in socio-technical networks to have a detrimental effect on design quality. However, Terceiro et al. [Terceiro2010] describe that modifications to the software carried out by developers that have a central place in socio-technical networks (compared to developers that have a more peripheral place in the socio-technical network) have a beneficial effect on design quality.

Development Effort (8%) measures the amount of development effort spent on software quality assurance [Herbsleb2006]. For instance, Hulkko et al. [Hulkko2005] describe that even though pair programming practices exhibit increased overal development effort, a large portion

of that increased effort translates into a larger documentation ratio of the source code, and thus to increased quality. Herbsleb et al. [Herbsleb2006] find that collaboration and coordination requirements of developers have a larger impact on development effort than any of the technical aspects captured in their study. Barbagallo et al. [Barbagallo2008] find that projects with a high centrality across their socio-technical developer networks are able to attract more external contributions to the open source projects under study, lowering the overall development effort across the project.

Deviations from Coding Standards (5%) is a software quality measure used exclusively in early studies encountered in this literature review. Deviations from standards are perceived as a negative aspect of software quality [Hulkko2005; Spinellis2006]. Hulkko et al. [Hulkko2005] describe that pair-programming practices result in more deviations from coding standards when compared to solo-programming practices. Spinellis et al. [Spinellis2006] report that they find no statistically significant relationship between geographic dispersion of development teams and deviations from coding standards.

Documentation Ratio (3%) captures the proportion of source code that is documented by source code comments. Only a single study [Hulkko2005] reports on documentation ratio as a software quality measure.

Expert Judgement (3%) defines a subjective measure of software quality based on expert opinion. The only study captured in this literature review that reported expert judgement as a software quality measure was carried out in the academic domain [Amrit2005], and in particular the quality of software deliverables of a student project is judged by the instructor grading the project.

Productivity (22%) captures an indirect software quality measure and is commonly expressed

as the amount of lines of code changed by a developer or development team per time unit. Within the context of software quality, productivity has a dual nature: on one hand the frequency of source code changes has been demonstrated to strongly correlate with defect density [Nagappan2008] (decreasing software quality), but on the other hand, increased productivity allows developers to carry out more corrective maintenance tasks (potentially increasing software quality).

The studies captured in this literature review investigate the impact of a variety of socio-technical observations on productivity. For instance, Bird et al. [Bird2006a] demonstrate that developers with high centrality in a developer communication network are the most productive. Ramasubbu et al. [Ramasubbu2011] provide empirical evidence that the organization of distributed development teams with respect to balancing team size and experience impacts productivity. Spinellis et al. [Spinellis2006] find that geographic distance between developers has only a small impact on overall development productivity. Cataldo et al. [Cataldo2008b; Cataldo2008a] demonstrate that communication and coordination requirements between developers negatively impact productivity.

Project Health (3%) is used in a single study captured by this literature review [Amrit2010], and indirectly captures software quality in an open source setting. Amrit et al. define project health with respect to the conformity of the open source project to the Onion model [Amrit2010].

Profit (3%) is used in a single study to capture the success of the software with the implicit assumption that a low quality piece of software might not be profitable due to low customer satisfaction and high corrective maintenance costs [Ramasubbu2011].

Time to Implement (16%) is an indirect software quality measures, similarly to productivity. This measure captures the overall amount of time that development efforts required to arrive at a

failure free software modification. Time to implement is a direct quality measure in a few studies of studies encountered [Herbsleb2003; Cataldo2006] and is directly related to the amount of modification requests for corrective maintenance that can be completed in the stabilization phase of the software development before the final release.

> *We observe that defect density is the most frequently described quality metric in the surveyed literature. However, we believe that while defects are a primary concern, future research could benefit from studying additional aspects of software quality, such as having a functional build, a strong design, as well as clean and documented code that eases future corrective and perfective software development efforts.*

2.4.4 *RQ4. What are the strategies, tools and data sources used to obtain socio-technical information about a software project?*

Data Sources

Table 2.11 summarizes our findings on the data sources that are used to extract socio-technical information about software development efforts. Overall, we found the following data sources:

- **Email Repositories (19%)**, such as Mailing Lists, record communication between developers, users, and other stakeholders of a software development effort [Bacchelli2010a]. The studies captured in this literature provide a strong case for email repositories as a major source of socio-technical information about communication and coordination concerns.
- **Chat Logs (16%)** such as IRC chat and Jabber chat record informal synchronous communication between developers [Cataldo2006] and provide socio-technical information about communication and coordination networks between developers.
- **Version Control Systems (86%)** store the source code and historical records of the source code together with metadata about changes between individual revisions of the

Table 2.11: Dissemination Matrix: Data Sources of Socio-Technical Information (RQ4)

Reference	Chat Logs	Email Repositories	MR Repositories	Qualitative Data	Misc. Data	Version Control System
[Mockus2000c]			✓			✓
[Herbsleb2003]			✓		✓	✓
[Amrit2005]				✓	✓	
[Hulkko2005]				✓	✓	
[Bird2006a]		✓				✓
[Cataldo2006]	✓		✓			✓
[Herbsleb2006]			✓			✓
[Spinellis2006]			✓			✓
[Espinosa2007]			✓			✓
[Ramasubbu2007]					✓	
[Weyuker2007]						✓
[Barbagallo2008]			✓	✓		✓
[Cataldo2008a]	✓		✓			
[Cataldo2008b]	✓	✓	✓	✓		✓
[Meneely2008]						✓
[Nagappan2008]					✓	✓
[Nguyen2008]	✓	✓	✓			✓
[Pinzger2008]						✓
[Barbagallo2009]			✓			✓
[Bird2009a]			✓			✓
[Bird2009c]			✓			✓
[Cataldo2009a]				✓		
[Hossain2009]			✓			
[Meneely2009]			✓			✓
[Wolf2009]	✓	✓	✓			✓
[Amrit2010]			✓			✓
[Bacchelli2010a]		✓	✓			✓
[Bettenburg2010a]			✓			✓
[Meneely2010]			✓			✓
[Terceiro2010]						✓
[Bicer2011]			✓			✓
[Canfora2011]		✓	✓			✓
[Cataldo2011]			✓	✓	✓	✓
[Eyolfson2011]			✓			✓
[Kwan2011]	✓	✓	✓			✓
[Ramasubbu2011]			✓	✓	✓	✓
[Shin2011]						✓
[Bird2012]			✓	✓		✓

source code, such as change author and change log messages [Mockus2000c]. Version control systems are the main source of technical information such as change size, and frequency [Mockus2000c], as well as CK metrics. But version control systems have also been demonstrated as a valuable source of socio-technical network information such as file-dependency networks [Cataldo2008b; Herbsleb2006; Bird2009a], and file-developer networks [Cataldo2008b; Meneely2010], as well as source of communication about change concerns [Eyolfson2011].

- **Modification Request Repositories (73%)** including bug tracking and task databases such as BugZilla track requests for modification of the source code, as well as progress on these requests [Mockus2000c]. In addition to providing a source for the measurement of the defect density of a software project [Mockus2000c], modification request repositories are a main source of socio-technical information, such as communication [Bettenburg2010a], task-developer networks [Hossain2009], or coordination requirements [Cataldo2006].

- **Qualitative Data (22%)** gathered through interviews and surveys [Amrit2005], is used for the building of hypotheses [Amrit2005], and corroboration of findings and results with practitioners [Herbsleb2003].

- We also encountered **Miscellaneous Data (19%)** such as Excel tables and text documents [Herbsleb2003] as a frequent data source of studies in the industrial domain.

Domain

To judge the external validity of research, records of the specific domain(s) for which each study was carried out are highly valuable. Open-source software projects have become an increasingly popular source of data for empirical research. Open-source data is often readily available online and has very few restrictions on usage. Even though open-source data enables replication of research (as opposed to commercial data which often restricts researchers to very specific non-disclosure agreements), software development processes in open-source development have

many different characteristics compared to industrial development, hence findings from one domain may differ from the other. We found the following domains (summarized in Table 2.12) described in literature:

- Studies in the **academic domain (5%)** involve the investigation of hypotheses through the help of students and colleagues [Hulkko2005]. Students and scholars have often very different expertise, background and work practices from practitioners, and thus findings based on this domain alone need to be examined carefully, as findings might not generalize to other domains.

- Studies in the **open-source domain (51%)** have the benefit of openness of data, which opens these studies up for possible replication. However, open source software development is often carried out on volunteer-basis by developers who are distributed geographically and who are distributed across different time zones [Bird2012]. As such asynchronous communication becomes a major impact factor on development and coordination processes in the open source domain [Barbagallo2009].

- Studies in the **commercial domain (59%)** describe software engineering efforts carried out by professional development teams that are managed through an organizational hierarchy and follow well-defined development processes and practices. Communication in this domain has been reported to happen frequently on an informal basis in a face-to-face fashion [Cataldo2006], and as such communication and coordination records are often not readily available or incomplete, making them misleading to outsiders [Aranda2009].

Data Gathering and Extraction Methods

Research in empirical software engineering has employed a variety of different methods of data gathering. Table 2.13 summarizes the data gathering methods described in articles included in

Table 2.12: Dissemination Matrix: Study Domain

Reference	Academic	Commercial	Open Source
[Mockus2000c]		✓	
[Herbsleb2003]		✓	
[Amrit2005]	✓		
[Hulkko2005]	✓	✓	
[Bird2006a]			✓
[Cataldo2006]		✓	
[Herbsleb2006]		✓	
[Spinellis2006]			✓
[Espinosa2007]		✓	
[Ramasubbu2007]		✓	
[Weyuker2007]		✓	
[Barbagallo2008]			✓
[Cataldo2008a]		✓	
[Cataldo2008b]		✓	
[Meneely2008]		✓	
[Nagappan2008]		✓	
[Nguyen2008]		✓	✓
[Pinzger2008]		✓	
[Barbagallo2009]			✓
[Bird2009a]		✓	
[Bird2009c]		✓	✓
[Cataldo2009a]		✓	
[Hossain2009]			✓
[Meneely2009]			✓
[Wolf2009]		✓	✓
[Amrit2010]			✓
[Bacchelli2010a]			✓
[Bettenburg2010a]			✓
[Meneely2010]			✓
[Terceiro2010]			✓
[Bicer2011]		✓	✓
[Canfora2011]			✓
[Cataldo2011]		✓	
[Eyolfson2011]			✓
[Kwan2011]		✓	
[Ramasubbu2011]		✓	
[Shin2011]			✓
[Bird2012]			✓

our systematic review. In particular, we found the following three main data gathering methods described:

- **Data Mining (95%)** as an automated form of data gathering. While researchers and practitioners can collect large amounts of data through data mining, giving their statistical evaluations higher levels of precision and confidence, large amounts of data are harder to inspect, clean and use more computational resources.

- **Interviews (14%)** and **Surveys (5%)** commonly result in qualitative data that requires manual analysis steps. Both methods can also be used to corroborate findings with experts, making them highly valuable for external validation [Herbsleb2003].

- **Manual Data Gathering (22%)** might need to be performed when data mining approaches are not applicable, or data sources such as software repositories are not available. For instance, desired socio-technical information may be recorded in textual documents that are kept by developers [Amrit2005].

We observe that data mined from source code repositories and modification request tracking systems are the primary sources of socio-technical information described in literature. Further, we note that communication repositories such as mailing lists remain largely unexplored. ***We demonstrate in Chapter 3 of this thesis, that the extraction of communication data from email repositories is a challenging task, and we describe tools and techniques to make communication information stored in email repositories accessible for data mining and analysis.***

Table 2.13: Dissemination Matrix: Data Extraction and Gathering Methods

Reference	Data Mining	Interviews	Surveys	Manual
[Mockus2000c]	✓			
[Herbsleb2003]	✓		✓	
[Amrit2005]		✓		✓
[Hulkko2005]		✓		✓
[Bird2006a]	✓			
[Cataldo2006]	✓			
[Herbsleb2006]	✓			
[Spinellis2006]	✓			
[Espinosa2007]	✓			
[Ramasubbu2007]				
[Weyuker2007]	✓			
[Barbagallo2008]	✓		✓	
[Cataldo2008a]	✓			
[Cataldo2008b]	✓			
[Meneely2008]	✓			
[Nagappan2008]	✓			
[Nguyen2008]	✓			
[Pinzger2008]	✓			
[Barbagallo2009]	✓			
[Bird2009a]	✓			
[Bird2009c]	✓			
[Cataldo2009a]	✓	✓		
[Hossain2009]	✓			
[Meneely2009]	✓			✓
[Wolf2009]	✓			
[Amrit2010]	✓			
[Bacchelli2010a]	✓			
[Bettenburg2010a]	✓			
[Meneely2010]	✓			✓
[Terceiro2010]	✓			
[Bicer2011]	✓			
[Canfora2011]	✓			✓
[Cataldo2011]	✓			✓
[Eyolfson2011]	✓			
[Kwan2011]	✓			
[Ramasubbu2011]	✓	✓		✓
[Shin2011]	✓			
[Bird2012]	✓	✓		✓

Table 2.14: Dissemination Matrix: Methods used to investigate Socio-Technical Relationships

Reference	Correlation Analysis	Descriptive Statistics	Principal Component Analysis	Regression	Machine Learning Methods	Qualitative Analysis
[Mockus2000c]				✓		
[Hulkko2005]		✓				✓
[Herbsleb2006]		✓		✓		
[Spinellis2006]	✓					
[Weyuker2007]				✓		
[Meneely2008]				✓		
[Nagappan2008]			✓	✓		
[Pinzger2008]	✓			✓		
[Bird2009a]	✓	✓		✓		✓
[Bird2009c]			✓	✓		
[Cataldo2009a]	✓			✓		✓
[Hossain2009]				✓		
[Bacchelli2010a]	✓		✓	✓		
[Bettenburg2010a]	✓	✓		✓		
[Bicer2011]		✓			✓	
[Eyolfson2011]	✓	✓				
[Ramasubbu2011]	✓	✓		✓		✓
[Bird2012]		✓		✓		
[Herbsleb2003]		✓		✓		✓
[Amrit2005]		✓				✓
[Bird2006a]	✓	✓				
[Cataldo2006]		✓		✓		
[Espinosa2007]	✓	✓		✓		
[Ramasubbu2007]		✓		✓		
[Barbagallo2008]			✓	✓		
[Cataldo2008a]				✓		
[Cataldo2008b]				✓		
[Nguyen2008]	✓	✓				✓
[Barbagallo2009]				✓		
[Meneely2009]	✓	✓		✓		
[Wolf2009]	✓				✓	
[Amrit2010]		✓				✓
[Meneely2010]	✓	✓		✓		
[Terceiro2010]	✓	✓				
[Canfora2011]	✓	✓				
[Cataldo2011]	✓	✓		✓		
[Kwan2011]	✓			✓		
[Shin2011]	✓	✓			✓	

2.4.5 *RQ5. What methods are applied to study the relationships between social and technical information?*

Over the past decade, empirical software engineering has employed a variety of different techniques from many different related research areas such as Machine Learning, and Statistics. In order to identify the most popular techniques, their applicability to the problem set of modeling software quality by means of socio-technical information, and future research opportunities, we record the modeling techniques proposed by literature. Table 2.14 summarizes the methods used to study socio-technical relationships to software quality. In particular, we identified the following methods:

- **Correlation Analysis (51%)** is used as a basic method to quantify the correlation between individual socio-technical measures and software quality measures on the one hand, and the inter-correlations between multiple measures on the one hand.

- **Descriptive Statistics (59%)**, on their own, are not an appropriate method of studying the statistical validity of observations. However, they provide valuable insights into the applicability and validity of the statistical and machine learning techniques used in a study.

- **Principal Component Analysis (11%)** can be used either as a method for resolving multi-collinearities between metrics and for correlation analysis. Principal component analysis however has the major drawback that metrics become merged into orthogonal dimensions, and these merged dimensions often prevent meaningful interpretation of results with respect to the original metrics [Bettenburg2010a].

- **Regression Models (70%)** were the most popular method of investigation of relationships. Regression models are generally harder to design and construct compared to Machine Learning techniques, however regression techniques offer greater insight into the findings [Bettenburg2010a].

- **Machine Learning Techniques (8%)** such as Decision Trees or Bayesian Classifiers act as black-box models, i.e., investigations of how the model was derived and operates is often not readily supported [Bettenburg2010a] (we found Decision Trees to be the noteable exception).

- **Qualitative Analysis (22%)** describes any form of manual corroboration of findings, with either the underlying data, or practitioners. Qualitative analysis can be helpful to discover the rationale or explanations behind observations [Herbsleb2003].

2.4.6 *RQ6. What are the main research opportunities available for future work?*

To better judge the validity and generalizability of findings presented by the studies captured in this literature review and identify open search opportunities, we combine Table 2.10 and Table 2.12, and count the points of empirical evidence provided in each domain by quality metric. An overview is presented in Table 2.15.

Table 2.15: Empirical Evidence on Quality Concerns, by Study Domain. Dark shaded (red colored) cells denote research areas that we we believe would benefit from additional empirical results.

Quality Concern	Industrial Domain	Open Source Domain
Defect Density	12	8
Vulnerability	0	3
Build and Integration Failures	3	1
Communication Frequency	2	1
Design Quality	0	3
Development Effort	2	1
Deviations from Standards	1	1
Documentation Ratio	1	0
Expert Judgement	0	0
Productivity	5	3
Project Health	0	1
Profit	1	0
Time to Implement	5	1

Overall, we find that studies between socio-technical relationships and defect density are the

most frequent type of studies encountered in this literature review, and have been well-researched in both the open-source domain and the industrial domain.

We also observe that a number of quality concerns lack empirical studies and thus findings have low generalizability. In particular we find that we lack empirical evidence on the relationships between socio-technical concerns and Build and Integration Failures, Communication Frequency, Development Effort, Deviations from Coding Standards, Project Health, and Implementation Time in the Open Source Domain. Similarly, we find that we lack empirical evidence on the relationships between socio-technical concerns and Vulnerabilities, Deviations from Coding Standards, Documentation Ratios, and Profit in the Industrial Domain.
From our summative overview of study concerns presented earlier in Table 2.5 we identify the following research opportunities in the industrial domain:

- Studies to investigate the impact of coordination concerns on defect density.
- Studies that investigate the relationships between socio-technical concerns and cross-system maintenance activities.
- Studies that investigate the relationships between of socio-technical concerns and design quality.
- Studies that investigate the relationships between relationships between socio-technical concerns and software vulnerabilities.
- Studies that investigate the relationships between socio-technical concerns and project health.

Similarly, in the open source domain, we identify the following research opportunities:

- Studies to investigate the impact of distributed development on Development Time.
- Studies that investigate the relationships between distributed development and defect density.

- Studies that investigate the relationships between socio-technical concerns and design quality.

- Studies that investigate the relationships between distributed development and development productivity.

2.5 Conclusions

This chapter presents a systematic literature review on research that relates socio-technical information about software development efforts to software quality. Through this literature review, we identified and described work that is related to our thesis hypothesis. We identified and critically disseminated a total of 37 primary studies from a broad range of software engineering venues. We uncovered a total of nine distinct research topics that are covered in existing literature, studying a broad range of relationships between socio-technical concerns and software quality. In particular we found a total of 39 different technical, social, and socio-technical metrics described in literature, together with investigations on their relationships to a total of 13 different quality metrics.

Within each research topic, we presented meta-analyses that summarize and outline the main findings within each topic, and identify future research opportunities in the area. The main findings of our systematic literature review are highlighted in the following.

- Adding social and socio-technical information to traditional product and process metrics based models provides significant benefits with respect to explanatory power, and predictive performance of the models. These findings hold true across the modeling of different outcomes such as software defects, software vulnerabilities, and build, as well as integration failures.

- Socio-Technical information provides a more complete picture of the factors influencing the quality of a software product. In particular, the strong empirical evidence of the value that socio-technical information adds to models supports the hypothesis that software engineering is a highly social process.

- Socio-Technical aspects such as coordination, communication, and awareness are of major concerns in both co-localized, as well as distributed development settings. In particular, with respect to distributed development absolute geographic distance has been demonstrate to have only a minor impact on software quality, while the organization of developers into organizational hierarchies (in the case of industrial development) and core-periphery structures (in the case of open source development) greatly influences coordination effort, reduces awareness, and creates communication barriers, thus greatly impacting software quality.

- Approaches from social network analysis applied to socio-technical networks are a major tool to investigate social processes in software development and the impact of these processes on the quality of the end product. Literature provides empirical evidence that entities with high centrality and betweenness in these socio-technical networks describe key elements that are crucial for software quality assurance.

- Most importantly, we find that existing literature provides empirical validation of Conway's Law through the definition and investigation of a socio-technical congruence measure that describes how well the social and technical parts of a socio-technical network overlap.

Part II

Tools and Techniques for Mining Communication Data

As we have discussed in Chapter 2, past research on software quality and software defect prediction heavily relies on the mining of technical information about the software and the software development process, such as the number of lines of source code, dependencies between files, the rate and size of changes, or test coverage.

In this thesis, we take a different approach. Our main focus lies on the different aspects of communication between developers, in particular *what they* communicate about (the contents), *how* they communicate (length, frequency), and *who* they communicate with. However, most communication data exists in the form of *unstructured data* [Bettenburg2010b], i.e., a mixture of natural language text and pieces of technical information. To illustrate, consider the following message exchanged between two developers of the IBM Eclipse software:

> *"I'm not entirely sure this is a dup. Basically, changes ended up in I20040219 that really shouldn't have – a miscommunication. While this is likely caused by this problem, I'm not sure. Could you check this again on a nightly build >= 20040220, or next week's integration build? Then report back.... "* **– Eclipse #52557**[4]

This message contains natural language text describing a potential reason for a problem that was discovered by the recipient of this message, as well as technical pieces of information, in this case build numbers of working and broken releases of the software (marked in green color).

This part of our thesis presents our tools and techniques to mining communication data such as the above from communication repositories, which record developer communication during day-to-day development activities.

- **Chapter 3: Mining Communication Data from Email Repositories.** In this chapter, we discuss some of the pitfalls and perils when mining communication data from one type of communication repository: email archives. This chapter highlights the problems of mining communication data with off-the-shelf text mining methods, such as those proposed by the data mining and information retrieval research communities. These approaches make the implicit assumption that the text that is being analyzed exists as well-formed English

[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=52557

language text, such as the text that one would find in a newspaper article. We argue that this assumption does not hold when mining communication data between developers, and that we need to adapt existing off-the-shelf methods to the software engineering domain. As part of this chapter, we identify several mining challenges and propose potential solutions. Based on the research presented in this chapter, we developed a publicly available tool for mining communication data from email repositories.

- **Chapter 4: Mining Technical Information from Communication Data.** In this chapter, we present a lightweight approach for separating the different pieces of technical information, such as the build number in the earlier example, from natural language text. We have manually curated a number of discussion messages to create a benchmark for assessing the performance of our approach, and to enable comparability with future approaches. Based on this benchmark, we find that the lightweight approach, presented in this chapter, has a statistically significant increased performance (23% higher precision, and 16% better recall) over state-of-the-art approaches.

- **Chapter 5: Linking Communication Data to Source Code.** In this chapter, we argue that traditional techniques for linking communication data to the source code of a software (also referred to as *traceability links* in the research community), are not appropriate when we are attempting to know the parts of the source code about which developers communicate. We present a novel approach for linking communication data to the parts of the source code that is communicated about, and through a case study demonstrate that our approach establishes links that are conceptually different from state-of-the-art approaches, and ultimately more appropriate.

3

Mining Communication Data from Email Repositories

The Collins English Dictionary defines collaboration as "[...] working with each other to do a task and to achieve shared goals [Sinclair1998]." A central aspect of collaboration is the communication required to coordinate work. A recent study in the domain of software engineering shows that engineers spend up to 2 hours each day communicating across various channels (face-to-face, chat, email) to coordinate their software development efforts [Wu2003]. As a result, communication repositories, such as mailing lists, contain valuable information about the history of a software project. Research is starting to mine this information to support developers and maintainers of long-lived software projects. However, such information exists as unstructured data that needs special processing before it can be studied. In this chapter, we identify several challenges that arise when using off-the-shelf techniques for mining mailing list data. In addition, we evaluate the negative impact of several of these mining challenges on the final data and research results, and we propose solutions to effectively tackle these mining challenges. We have implemented a publicly available tool for mining mailing list repositories, which has already found extensive use in the research domain [Jiang2013; German2013b; Thomas2011].

3.1 Introduction

Electronic mail is an established form of communication in networked computing environments. Mailing list software distributes messages to a predefined list of recipients and is widely used in software development. There it aids day-to-day development and enables communication between project stakeholders, e.g., developers and users. Messages sent over these mailing lists contain a multitude of information on the project, such as important development decisions, discussions of the source code, and support requests. Software maintainers can use this information to study corrective activities [Weissgerber2008], developer communication [Rigby2007], or knowledge recovery [Cubranic2003].

Although mailing list data is often readily available online, transforming the data into a structured format that is suitable for subsequent analysis is a challenging task. Messages are often stored in email archives and need to be extracted before they can be used. However, mailing list archives contain duplicate and invalid data, stored in raw formats, which need further processing. Additionally, up to 98.4% of electronic messages contain noise that threatens the applicability of text mining approaches [Tang2005]. Researchers need to be aware of potential pitfalls and take special care before using the information mined from mailing list archives.

In this chapter, we identify difficulties that arise when processing mailing list data. These difficulties are present in most stages of the mining process, such as data collection, data extraction and information processing. Previous research has anecdotically noted the presence of several challenges, but documented them only loosely, as they are a by-product of the research work conducted, rather than the main scope. Mining raw mailing list data yields potential risks to the accuracy of research results and should be avoided.

The difficulties with mining mailing list data and the solutions described in this chapter, generalize to a large extent to other types of recorded communication data, such as chat-logs [Shihab2009b; Shihab2009a], discussion threads attached to issue reports [Bettenburg2010a], or code reviews [Bettenburg2013a].

3.1.1 Contributions

The work presented in this chapter makes the following contributions to the research area: first, we present a holistic summary of the pitfalls and perils of mining mailing list data, when using off-the-shelf tools. Second, we present solutions to the described problems.

3.1.2 Organization of this Chapter

This chapter is organized as follows. In Section 2 we highlight the risks of using unclean communication data through example data analysis tasks. In Section 3 we present challenges that arise when using off-the-shelf techniques for mining of communication data from email repositories. We present the work related to our study in Section 4 and conclude this chapter in Section 5.

3.2 Background

Electronic messages and related technologies are specified in *Request For Comment* documents (RFCs), published by the Internet Engineering Task Force (IETF). Messages consist of two parts: a collection of email headers and the content.

Headers contain various information such as the message *sender* and *receiver*, the time and date at which the message was sent and *routing information* about the path the message has taken over the network. The headers also include a *subject* to indicate the message's motivation and contents. In addition to the minimum set of headers every message must contain, users can add custom headers for their own purposes.

The *content* part contains the actual message. The initial RFC for the electronic message format restricted content to the exclusive usage of plain text. As markup languages became popular, extensions to the email format allowed for multi-format messages, like *HTML emails*.

While plain text messages have the advantage of maximizing compatibility, multi-format emails offer the ability to stylize the text and store extra information.

These enhancements over the original mail format are specified by the Multipurpose Internet Mail Extensions (*MIME*) standard [Freed1996b; Freed1996c; Moore1996; Freed1996d; Freed1996a] giving this type of electronic mail documents the name *MIME-messages*.

Besides different formats, the MIME standard also specifies *character encodings* and *attachments*. Character encodings describe sets of characters and how these characters are represented in a binary form. Such character encodings are important, as different languages may contain language-specific characters as part of their writing systems.

With attachments, users can transmit additional non-text data together with their messages, e.g., documents like spreadsheets, or electronic fingerprints that can be used to digitally sign a message and prevent alteration of its contents.

3.2.1 *Motivating Examples*

Summaries of recent developer discussions can be useful for decision makers to monitor the development progress and to identify topics of high interest, to recover knowledge about design decisions, an to aid the maintenance of legacy systems. For instance, we use the contents of developer discussions in Chapter 6 to understand the relation between the different aspects of developer communication and software quality.

Communication data, such as email discussions, is stored in a textual way that humans can easily read and understand. However, using this data as-is in computerized, content-based analyses, yields hidden, yet severe risks for the validity of the obtained results. In this section, we present two examples to highlight the risks of using communication data that has not undergone the needed pre-processing steps that make it suited for machine-consumption.

Example 1 : Analysis of Contents

> "What were the hotspots our developers discussed the most in the past few days?"

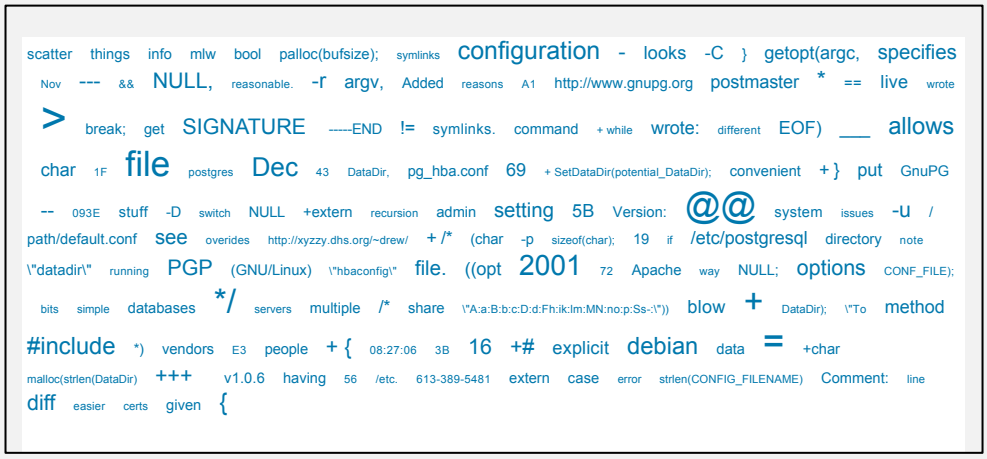
In this example we use tag clouds, a concept from information retrieval, to visualize the contents of a discussion thread. Tag clouds display the most frequent terms weighted by font size and color. The larger and more visible a term is presented in a tag cloud, the higher its semantic value for the text. Tag clouds share this concept with many information retrieval and machine learning algorithms, e.g., text classification, which also operate based on most frequently appearing terms. *Through these properties, tag clouds directly visualize the quality of the input data that we would use in subsequent information retrieval and data mining approaches for knowledge discovery and experimentation.*

Figure 3.1 shows two tag clouds visualizing the contents of the same discussion thread on the PostgreSQL mailing list with the topic *"Explicit config patch 7.2B4"*, starting at December 16th, 2001. This discussion centers around the possibility of passing command line arguments to the PostgreSQL server executable, which allow the user to specify the locations of the server's configuration files, because many Linux distributions, besides Debian, scatter configuration files around in the file system.

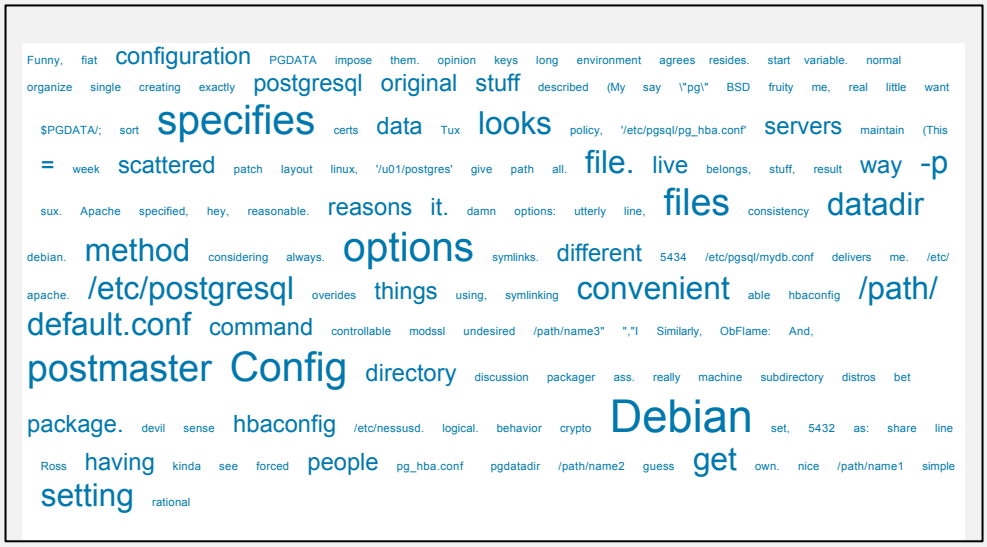
The discussion includes email messages that contain non-natural language text parts, such as source code fragments that demonstrate how to realize these command line options, patches to implement this new feature in the source code base and digital signatures of the authors. Both tag clouds summarize the same discussion.

The first cloud, presented in Figure 3.1a, is generated using the contents of the email messages that form the discussion thread as-is, i.e., without prior processing of the message bodies. The second cloud, presented in Figure 3.1b, is generated from the same email messages, however the messages were cleaned up significantly by removing attachments, signatures and quotations, as well as transforming all remaining parts into English language text.

Comparing both tag clouds, we can see that the tag cloud generated from uncleaned mailing



(a) Tag cloud generated from unprocessed communication data.



(b) Tag cloud generated from processed communication data.

Figure 3.1: Summarizing the contents of the same discussion thread using tag clouds generated from as-is (a) and processed (b) communication data.

list data contains a large amount of noise, which renders the interpretation of the discussion's contents a challenge. On the opposite, the summary produced from the cleaned discussion thread is much more helpful in giving a good idea of the contents of the discussion.

Example 2: Analysis of Social Structures

> "Who is the developer performing the most user support?"

Information on a person's contributions to the project and his rate of activity can be of interest to decision makers, e.g., to identify those developers that are central to processing user support requests.

When trying to relate the mailing list identities of developers to project activity, we encounter the problem of multiple aliases: the same person can use multiple identities, for instance due to different identities depending on his current workplace. When measuring the activity of such a developer without resolving his multiple identities first, we obtain a measure which is lower than his real activity.

In this example we consider the contribution of Alvaro Herrera to the developer mailing list of the PostgreSQL project. Over the course of several years, he has used six different email addresses to sent messages from. From his most frequently used identity he contributed 1,975 discussion messages, but also 1,172 messages from his other five identities. Alvaro is among the top contributors to the project, however when determining the top ten most active developers, and considering only his main identity, we would ignore about 40% of his activity and not consider him in this group.

3.3 *Study Design*

In order to systematically identify and evaluate the impact of the challenges that arise when using off-the-shelf techniques for processing mailing list data, we perform a series of case studies. Where off-the-shelf techniques are not available for evaluating the discussed challenge, we implemented custom heuristics based on ideas proposed in previous literature.

For the study presented in this work, we use the mailing lists development-specific mailing lists of 22 projects that belong to the GNOME Desktop environment [GNOME2009] and the PostgreSQL database system [PostgreSQL2009]. The data for these mailing lists is available online in the form of multiple compressed mail archive files, each containing a month worth of discussion data. For each project we collected all data available from the start of each mailing list until December 31, 2008. Overall, these mail lists contain more than 450,000 messages.

3.3.1 *Evaluation Process*

Many research fields, such as information retrieval and text-mining, maintain reference data sets, often referred to as *"gold standards"*. These data sets consist of manually inspected and labeled data and are used to advance methods and techniques of the community based on common standards. Such reference data ensures that research results can be reproduced and that methods are comparable. However, no such gold standards exist for the use of mailing lists in software maintenance, although the importance of reference data sets and benchmarks for the software engineering community has been noted in past research [Sim2003]. Due to the lack of reference data sets and the in-feasibility of a manually inspecting of all the mailing list data available to us, we need to resort to statistical techniques for our quantitative analysis.

For each analysis performed in this chapter, we chose random samples from all available messages for further manual investigation. When taking a sufficiently large amount of random samples, we can estimate the properties of the whole data set. The amount of samples needed can be calculated based on three pieces of information: the size of the complete data set, the percentage of error we allow and the size of error margin.

Since a random sample is usually not completely representative of the whole data from which it was drawn, findings based on that random sample involve a random variation called *sampling error*. This error can be described in terms of a *confidence level* and a *confidence interval*. The confidence level is a measure on the likelihood, expressed as a percentage, that the obtained

results are real and repeatable, and not produced by random. The confidence interval describes a margin of error, as a percentage, of the obtained results. In practice, sampling error is usually stated at a 95% confidence level, with a 5% confidence interval.

For instance, if we have a data set containing 200,000 data points, we need to inspect a random sample of 383 points to be able to extrapolate findings with a confidence of 95% and allow for and error of our findings of ±5%. For example, if we find 268 messages (70%) to be flawed, we can be 95% confident, that between 130,000 (65%) and 150,000 (75%) of all the messages are flawed in the same way. In the rest of this paper we refer to this type of analysis as a *statistical evaluation* of our findings.

As the amount of messages for each of the studied mailing lists varies greatly, we have to be careful when drawing a random sample. When sampling, we are more likely to draw messages from mailing lists that contain lots of messages. In order to counter this possible bias, we employ a *stratified random sampling* technique, to take a proportion of messages from each of the 23 mailing lists.

3.4 *Processing Mailing List Data with Off-the-Shelf Techniques*

Name of Challenge	**Level of Automation**	**Impact on Quality**
Message Extraction	Automated	low
Duplicate Removal	Automated	high
Language Support	Automated	medium
MIME/Attachments	Automated	high
Quotes/Signatures	Semi-Automated	high
Thread Reconstruction	Semi-Automated	high
Resolving Identities	Semi-Automated	high

Table 3.1: Overview of challenges presented.

In this section, we discuss challenges with using off-the-shelf techniques for mining mailing list data. An overview is presented in Table 3.1. For each challenge we assign a notion of automation and impact on data quality. Some challenges presented cannot be addressed in a completely automated manner and need manual tuning before reliable results can be obtained. We denote these as semi-automated mining challenges. From a combination of manual efforts required to tackled each challenge, and the impact on the data quality, we gain an intuition of the overall severity associated to each challenge.

3.4.1 *Extracting Messages*

Many open-source software projects store the messages of their mailing lists in *mbox* files [Robles2009], which represent textual databases that contain linear sequences of electronic messages. These messages need to be extracted before they can be analyzed. However, the extraction process requires knowledge about the structure of the archive. Additionally, competing MBOX specifications disagree on the format of the mail archive. Both the performance of extraction tools and the deficiencies of erroneous mailing list archives have an immediate impact on the quality and quantity of the extracted data.

As opposed to common internet standards like name resolution and data transport, the storage of electronic messages was never formally specified. As a result, implementation details on MBOX storage specifications are not well documented. First, for proprietary formats used by commercial applications like LOTUS Notes [IBM2009] and MICROSOFT Exchange [Microsoft2009a], public specifications are rarely available. Second, for open-source formats, like the MBOX format, multiple competing specifications exist, that are not compatible with each other. Hence, it is important to first identify the format of the archive files being used and to use an appropriate extraction mechanism. Additionally the chosen extraction mechanism needs to be robust enough to account for the errors present in most mail archives.

In a first case study on using off-the-shelf extraction tools for communication repositories

stored in the MBOX format, we used the following four popular and publicly available tools to extract messages from the mail archives of all 23 projects:

1. MOZILLA **Thunderbird** is an open-source email client with support for MBOX files.

2. PERL **Mail::Box framework** [Overmeer2002] is an API framework for processing email in the PERL programming language.

3. PYTHON **email** is an email framework distributed with the PYTHON programming language.

4. UNIX **formail** is part of the procmail mail processing package on UNIX-like operating systems where it acts as a mail filter and re-formatting tool.

Table 3.2 presents our findings on the amount of messages extracted using these four tools and their level of agreement. We use the average of the normalized distances from the mean values as a metric for agreement. Higher values of agreement signify that all methods extracted a similar amount of messages.

Overall, the results indicate a high level of agreement in the number of messages extracted for most mailing lists. However, we observe that none of the four tools extract the same amount of messages across all 23 mailing . For instance, for the GNOME libsoup development mailing list, formail greatly disagrees with the rest of the tools on the amount of messages present in the mailing list. Further analysis reveals that disagreement is largely due to implicit requirements made by the extractors on the MBOX specification details that they follow. For example, during the detailed evaluation of our results, we find that some mail archives do not separate subsequent messages with an empty line. As a result, some tools such as the UNIX `formail` tool considers them as attachments, rather than separate messages.

	Number of messages extracted using				
Mailing List	**ThunderBird**	**Mail::Box**	**Python Mail**	**Unix Formail**	**Φ**
pgsql-hackers	168,801	168,359	168,929	168,317	0.98
deskbar-applet	2,795	2,795	2,795	2,199	0.90
ekiga	8,983	8,981	8,983	6,449	0.90
eog	1,399	1,399	1,399	1,258	0.91
epiphany	12,431	12,426	12,432	11,013	0.91
evince	3,127	3,126	3,127	2,420	0.90
evolution	153,912	153,636	153,961	150,449	0.92
games	3,995	3,992	3,995	3,192	0.91
gdm	6,135	6,135	6,135	5,229	0.91
gedit	5,082	5,083	5,086	4,374	0.90
gnomecc	3,370	3,364	3,370	2,878	0.91
libsoup	173	172	173	83	0.87
metacity-devel	732	732	732	602	0.91
multimedia	5,900	5,897	5,900	5,583	0.91
nautilus	58,241	58,123	58,271	55,910	0.91
network	2,544	2,544	2,544	2,530	0.90
orca	17,213	17,205	17,214	12,152	0.90
power-manager	2,370	2,370	2,370	1,998	0.91
screensaver	352	352	352	276	0.90
seahorse	306	305	306	151	0.87
system-tools	2,561	2,561	2,561	2,397	0.93
themes	3,718	3,717	3,718	3,422	0.91
utils	1,194	1,194	1,194	1,025	0.91
Φ=Level of Agreement					

Table 3.2: Messages extracted from GNOME and PG mailing list archives using different tools.

3.4.2 *Removing Duplicates*

One essential part of any cleaning process in data mining involves the identification and removal of duplicate data [Lee1999]. This step is of utmost importance when the mined data is used in aggregation functions or frequency analyses. Duplicate entries will result in false or potentially misleading results. The 3 main sources of duplicate messages on mailing lists are:

1. *Network problems*, i.e., timeouts, can cause a message to be sent multiple times.

2. *Software errors* in the mailing list software can cause messages to be recorded multiple times.

3. *Accidental resubmission* (e.g., a user clicked a "send" button multiple times) can also result in duplicate messages to be transferred to the mailing list.

Solutions to this challenge, e.g., similarity measures like hashing or near-miss identification, can easily be automated.

	# of Messages		
List	**Unique**	**Duplicates**	**Ratio**
pgsql-hackers	168,267	87	0.05%
deskbar-applet	1,117	1,678	150.22%
ekiga	5,407	3,574	66.10%
eog	505	894	177.03%
epiphany	5,748	6,678	116.18%
evince	1,371	1,755	128.01%
evolution	54,158	99,457	183.64%
games	1,598	2,394	149.81%
gdm	2,595	3,540	136.42%
gedit	2,258	2,825	125.11%
gnomecc	1,499	1,865	124.42%
libsoup	110	62	56.36%
metacity-devel	282	450	159.57%
multimedia	1,698	4,199	247.29%
nautilus	22,516	35,607	158.14%
network	703	1,841	261.88%
orca	11,954	5,251	43.93%
power-manager	1,068	1,302	121.91%
screensaver	146	206	141.10%
seahorse	274	31	11.31%
system-tools	1,930	631	32.69%
themes	1,392	2,325	167.03%
utils	418	776	185.65%

Table 3.3: Ratio of duplicated messages encountered in GNOME and PG mailing lists

In a case study, we want to evaluate the amount of duplicate messages in mailing lists. In order to identify duplicate messages in the mailing list data, we use a cryptographic hash function on the contents of each message.

Table 3.3 presents our findings on the ratio of unique messages to duplicate messages stored in the mailing list data of each project. Whereas the PG mailing list shows only a small ratio of duplicated entries of 0.05 percent, some mailing lists exhibit much higher ratios of up to 261.88%. During the statistical evaluation of our results, we find that messages in certain mailing list are successively stored multiple times in the mail archives, probably due to an error in the mailing list software when generating the archive files.

Our case study highlights the importance of identifying and removing duplicate messages from mailing lists. A hashing approach can help to identify exact copies of the data. To identify near-copies of messages, i.e., messages that differ only in a tiny fraction of text, we can adapt techniques from text-retrieval and web-search communities [Galhardas2000; Monge2000].

3.4.3 *Handling Multiple Languages*

Since geographically distributed software development is increasing in both open-source [Herraiz2006] and industry [Herbsleb2001], mailing lists are used for communication of a multitude of developers with different cultural backgrounds and languages. Character encodings specify how text in the writing systems of different languages is represented in a binary form [Whistler2008]. Problems arise when the encoding of a message is ignored during the data mining process. For instance, the name “Réné” encoded in a French character set would be transformed to “Rn” when treated as English text. In order to safeguard the mined information from data loss, it is important to determine the appropriate encoding and safely translate text to an encoding like Unicode, which can handle multiple languages simultaneously.

Existing internationalization solutions like the MOZILLA character encoding detection algorithm [Li2001] can be used to robustly unify multi-language archives automatically. Thus, messages that specify an encoding different from the main language of the mailing list – or

no encoding at all – need to be safely converted to a common format like Unicode, to prevent potential data loss.

In a case study, we use the `MOZILLA` character encoding detection algorithm [Li2001] to quantify the amount of mailing list messages in non-native formats. This algorithm combines knowledge about the possible characters in different languages, frequencies of words and 2-char sequence distributions to detect the language and encoding used for a text. We consider English to be the main language (SE) for communication on all mailing lists studied and record the amount of messages that miss to specify any particular encoding or denote an invalid encoding (NSE) that does not match its contents.

	# of Messages			
List	**SE**	**NSE**	**Total**	$\frac{\text{NSE}}{\text{Total}}$
pgsql-hackers	107,884	60,383	168,267	35.89%
deskbar-applet	467	650	1,117	58.19%
ekiga	3,758	1,649	5,407	30.50%
eog	305	200	505	39.60%
epiphany	2,546	3,202	5,748	55.71%
evince	579	792	1,371	57.77%
evolution	11,748	42,410	54,158	78.31%
games	1,043	555	1,598	34.73%
gdm	1,948	647	2,595	24.93%
gedit	1,048	1,210	2,258	53.59%
gnomecc	765	734	1,499	48.97%
libsoup	77	33	110	30.00%
metacity-devel	194	88	282	31.21%
multimedia	730	968	1,698	57.01%
nautilus	10,935	11,581	22,516	51.43%
network	195	508	703	72.26%
orca	8,202	3,752	11,954	31.39%
power-manager	420	648	1,068	60.67%
screensaver	86	60	146	41.10%
seahorse	222	52	274	18.98%
system-tools	987	943	1,930	48.86%
themes	569	823	1,392	59.12%
utils	147	271	418	64.83%

SE = standard encoding (US-ASCII and UTF-8)

NSE = non-standard or unspecified encoding

Table 3.4: Amount of messages with standard and non-standard character encodings.

The findings of our case study are presented in Table 3.4. The proportion of messages that need conversion ranges between 18,98% for the GNOME seahorse list, to up to 78,31% for the GNOME evolution mailing list.

> *Our results reveal that a substantial amount of messages is present in either non-native or unspecified character encodings. In order to safeguard the mined information from data loss, it is important to determine the appropriate encoding and safely translate text to an encoding like Unicode, which can handle multiple languages simultaneously.*

3.4.4 *Handling MIME Messages and Attachments*

The initial RFC for the email format restricted content to the exclusive usage of plain text. As markup languages became popular, extensions to the email format allowed for multi-format messages, like *HTML emails*. The enhancements over the original mail format are specified by the Multipurpose Internet Mail Extensions (*MIME*) standard [Freed1996b; Freed1996c; Moore1996; Freed1996d; Freed1996a] giving this type of electronic mail documents the name *MIME-messages*. that can be used to digitally sign a message and prevent alteration of its contents.

While plain text emails have the advantage of maximizing compatibility, MIME messages offer the ability to stylize the text and store extra information. Additionally to formatting, the MIME standard also specifies *attachments*. With attachments, users can transmit additional non-text data together with their messages, e.g., documents like spreadsheets, or electronic fingerprints.

While the MIME extension allows for specialized and custom styled messages that can contain binary data, it comes at the cost of making the data mining approach more challenging, as extra mechanisms are needed to handle these contents. A number of different composite MIME-types exist, that have implicit semantics for the message contents [Freed1996c].

- The **multipart/mixed** composite type contains independent body parts in a particular order. An electronic message with a text part and an attached document is an example of this type. To handle this type of composite format correctly, we need to consider only the first part as the main content of the message and additional parts to contain separate information.

- The **multipart/alternative** composite type contains body parts that are alternative versions of the same information. Parts are ordered in increasing order of preference. An example for this type would be an email containing stylized text in the html format and the same information as plain text. To handle this format correctly, we need to consider only the first part of the message. Additional parts contain the same information, but in order of decreasing format simplicity.

- **Multipart/signed** messages contain the main contents of the messages in the first part. The second part is a cryptographic token, which can be used to verify the identity of the message sender or to protect the contents of the message from alteration. To handle messages in this format correctly, we need to consider only the first part as the main content of the message.

- In **multipart/related** types, all parts are chunks of the body. Their aggregation forms the whole body. One example would be an html email containing pictures. To handle this type correctly, we need to aggregate the information of all parts that are in a textual to form the main contents of the message.

In our case study we are interested in determining the proportion of mailing list messages that are in a format other than plain text, since the contents of these messages need to be treated differently. We use the Java Mail API framework to parse all mailing list messages and record their MIME types. We find only a fraction of 0.01% of messages to be in rich-text format. However, we discover that 3.36% of all messages are HTML emails and that 6,31% of

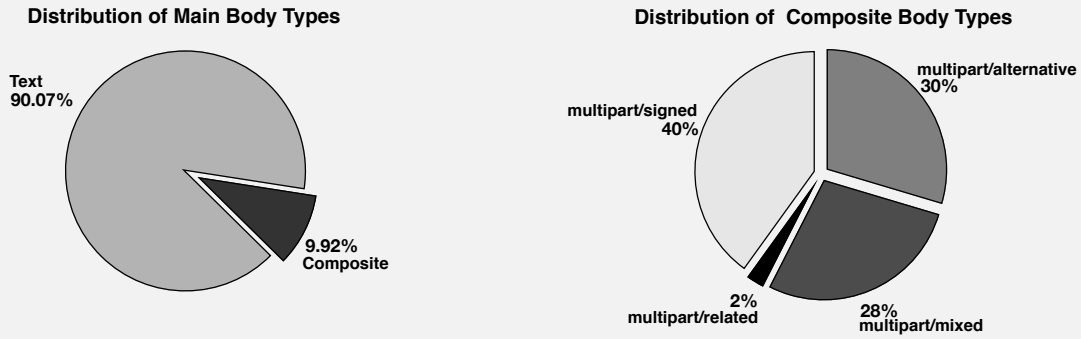


Figure 3.2: Distribution of the main message body types.

all messages contain one or more binary attachments.

Figure 3.2 presents our findings on the distribution of the main body types for all messages. Overall, about 90 percent of the message main bodies consist of simple textual formats like plain text, html or rich text. The remaining ten percent of messages are of composite types: 40% are digitally signed messages (multipart/signed), 30% have bodies with alternative versions of the same information (multipart/alternative) and 28% of the composite-type messages have multiple independent parts (multipart/mixed). Only 2% of the composite-type messages contain related contents (multipart/related).

Our results show that about ten percent of mailing list messages need special attention when processing mailing list data. Stylized messages need to be translated from HTML or Rich Text to a plain text format, attachments need to be separated from the messages' contents and stored separately, and composite type messages need to be handled according to their implicit semantics.

3.4.5 *Removing Quotes and Signatures*

In mailing list discussions, users usually refer to text from a previous participant by quoting parts of the original message to give more context and meaning to their contributions, as S.

Hambridge from Intel states in his network etiquette guidelines [Hambridge1995]:

"If you are sending a reply to a message or a posting be sure you summarize the original at the top of the message, or include just enough text of the original to give a context."

However, the additional text might not be desirable for text- and data mining approaches, as it is redundant information that has already been encountered before. Quoted text typically begins with one or more ">" signs at the beginning of a line - one for each level of quotation - and is easily removed automatically.

Signatures are typically used as a "soft" proof of identity and to indicate that no more text is following in a message [Hambridge1995]. Signatures contain a variety of artifacts, such as contact information, text graphics, famous quotes and trivia. For instance, `Google Mail` [Google2009] can be set up to include a random quote as a signature when sending email. Many free email services also add advertisements as a signature when a message is sent. As a result, the information in a signature block is often repetitive and unrelated to the message. Figure 3.3 illustrates a sample signature from a participant on the PG developer's mailing list.

```
                                                  _________________________
.  .  _  ._  . .   .__    .  . ._. .__ .   . . .__  | Neighbourhood Coder
|\/| |_| |_| |/    |_     |\/|  |  |_  |   |/  |_   |
|  | | | | \ | \   |__ .  |  | .|. |__ |__ | \ |__  | Ottawa, Ontario, Canada
One ring to rule them all, one ring to find them, one ring to bring them all
and in the darkness bind them...
```

Figure 3.3: Sample signature extracted from a message on the PG mailing list.

Current solutions to this challenge exist only as semi-automated tools or processes that need to resort to manual inspection and fine-tuning of parameters to yield good results.

In order to quantify the number of messages with signatures perform a statistical evaluation on message bodies for each mailing list and extrapolate from these results. We estimate a considerable amount (81,43%) of messages to contain a signature. As we know of no existing off-the-shelf tool to remove signatures from email messages, we implemented a prototype of an

algorithm to evaluate the chances for success of an automated signature removal method. The algorithm takes a small sample of 5 consecutive messages, ordered by sending date ,from each sender and identifies all those lines that are common to all five messages. Common lines at the end of the message contents are considered to comprise the signature of the message sender. In order to account for signatures changing over time, we run this process multiple times for each sender, at different dates. Again, we perform a statistical evaluation of the signature removal. We find that the heuristic fails to identify those signatures that change randomly each time a message is sent. Furthermore, a message sender would occasionally use the same salutations, e.g., "Cheers, John", at the end of five or more consecutive messages. Whether such salutations can be considered as signatures is subject to debate and depends on the focus of the textual analysis carried out on the mailing list data. On average, we achieve an accuracy of 76.81% which is in the performance range of machine-learning based approaches [Carvalho2004].

3.4.6 *Reconstructing Discussion Threads*

An email *discussion thread* is a set of messages that are logically related, e.g., multiple answers to a question. The messages in the set form a tree-shaped hierarchy. Whenever a user participates in the discussion, his message becomes a child of the message he is replying to. The initial message starting a new topic is the root node.

However, mailing list archives commonly store messages based on their temporal order rather than their logical grouping. As such, the hierarchical order has to be re-constructed after the messages have been extracted. The email standard specifies the `message-id` and `in-reply-to` header fields for this purpose [Crocker1982].

As an additional challenge, a user's email client is responsible for storing unique identification information on messages in a special header field. This header is optional, so in practice one cannot rely on threading information to be present. For instance, the `MICROSOFT` Outlook [Microsoft2009b] email client did not implement a message-id header until its latest

version (Outlook 2007).

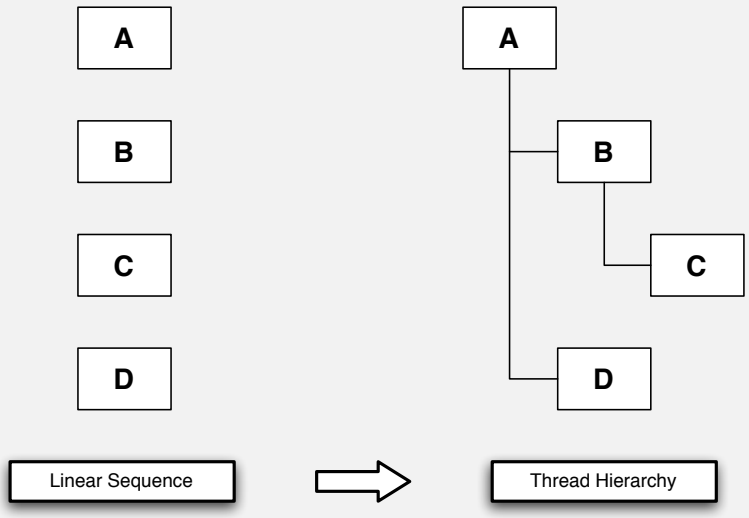


Figure 3.4: Hierarchical Information has to be reconstructed from a linear sequence of messages

However, there are two main drawbacks to this method. First, the sender of a message is responsible for obtaining a world-wide unique message id for each message sent. As there is no central authority issuing these ids, they are not necessarily unique in practice. Second, the message-id header is optional and one can not rely on this information to be present.

In order to make thread reconstruction more reliable and enable it for messages that are missing the message-id or in-reply-to headers, we can use heuristics based on the additional information available from messages.

First, the `references` header contains the message identifiers of "related correspondence". When replying to a message, email clients are supposed to copy the references from the parent message and append this message's id. Assuming that some clients specify correct identifiers, references can help to reconstruct the message hierarchy.

Second, the email `subject` is used to indicate the purpose and content of the message. Message subjects are often prefixed to signal actions taken to the message. For instance, many

	Heuristic					
List	**H1**	**H2**	**H3a**	**H3b**	**H3c**	**H3d**
pgsql-hackers	48,643	36,666	30,409	29,917	29,743	29,717
deskbar-applet	369	353	340	340	340	340
ekiga	1,442	1,392	1,232	1,207	1,200	1,200
eog	259	242	236	234	233	232
epiphany	1,852	1,730	1,619	1,608	1,608	1,607
evince	604	589	570	568	566	566
evolution	19,328	17,489	15,834	15,752	15,718	15,717
games	584	562	534	531	531	531
gdm	1,166	1,107	1,050	1,042	1,040	1,040
gedit	1,047	963	924	920	919	919
gnomecc	414	349	316	315	311	311
libsoup	43	42	41	41	41	41
metacity-devel	76	64	60	60	59	59
multimedia	569	535	507	507	507	507
nautilus	8,081	6,283	5,650	5,591	5,582	5,582
network	287	277	270	267	267	267
orca	4,554	3,868	3,612	3,599	3,598	3,597
power-manager	330	316	308	305	305	303
screensaver	31	31	31	31	30	30
seahorse	159	117	117	116	116	116
system-tools	1,002	827	795	792	792	792
themes	498	472	448	448	447	447
utils	290	285	280	280	279	279
H1 = using message-id						
H2 = using message-id and references						
H3a-H3d = using message-id, references and subjects						

Table 3.5: Total amount of threads reconstructed using different heuristics and parameters.

mailing lists prefix messages with the list name between square brackets, e.g., "`[Evolution] Some Subject`" and users often use the prefixes "`Re:`" for replies and "`Fwd:`" for forwarded messages. Thus, we can group related messages based on their subjects and establish a hierarchy using the prefix.

We performed a case study on the performance of thread reconstruction using different off-the-shelf techniques and present our results in Table 3.5. In experiment H1, we use only information available from the `message-id` and `in-reply-to` headers. In experiment H2, we use the unique ids, as well as information from the list of related correspondence. In experiments H3a to H3d, we use unique ids, related correspondence and message subjects. These four

experiments use different sizes of sliding windows when looking for related subjects: one week, four weeks, six months, one year. A larger windows yields the risk of accidentally associating messages with discussion threads that happen to have the same subject but are a long time apart (although that sometimes is correct).

The number of threads reconstructed using more information decreases when more information is used. This is due to the fact that we initially start by assuming every message on a list to be a root message. When identifying more messages to belong to a thread, and hence the number of threads decreases. Our statistical evaluation showed less than 7% of false positives, i.e., messages incorrectly associated with a thread.

> *Our results show that we can resolve up to 1.62 times the amount of threading information found by current techniques (*PG*), when using information from related correspondence and message subjects additional to the* `message-id` *and* `in-reply-to` *headers. when considering subject similarity, a sliding window of 4 weeks (H3b) seems to yield the best trade-off between the quantity of threads reconstructed and messages falsely associated.*

3.4.7 *Resolving Multiple Identities*

Some participants use multiple email addresses when taking part in discussions on a mailing list [Bird2006a]. These addresses are aliases for individual personalities and should be resolved before using the data. Ignoring this problem can lead to problems when doing quantitative and social analyses.

We know of no readily available tool for identification and merging of email aliases. Thus, we follow the idea presented by Robles et al. [Robles2005] and implemented a heuristic based on regular expressions to identify and merge multiple identities of mailing list participants. In a case study, we performed a merging of identities for each mailing list separately and present our results in Table 3.6. We display the number of participants before and after merging of aliases,

	# Identities		Difference	
List	**Before**	**After**	**Absolute**	**Relative**
pgsql-hackers	5,518	4,465	1,053	19.08%
deskbar-applet	104	102	2	1.92%
ekiga	665	648	17	2.56%
eog	122	113	9	7.38%
epiphany	921	842	79	8.58%
evince	416	400	16	3.85%
evolution	6,235	5,522	713	11.44%
games	178	169	9	5.06%
gdm	708	626	82	11.58%
gedit	553	514	39	7.05%
gnomecc	191	166	25	13.09%
libsoup	33	31	2	6.06%
metacity-devel	59	54	5	8.47%
multimedia	309	264	45	14.56%
nautilus	2,557	2,147	410	16.03%
network	114	96	18	15.79%
orca	502	448	54	10.76%
power-manager	197	190	7	3.55%
screensaver	26	24	2	7.69%
seahorse	38	35	3	7.89%
system-tools	363	288	75	20.66%
themes	276	236	40	14.49%
utils	137	111	26	18.98%

Table 3.6: Amount of different mailing-list identities before and after merging aliases.

as well as the absolute amount of identities that were merged, and the overall proportion of identities that were affected.

Our results show that between 1.92% (deskbar-applet) and 20.66% (system-tools) of the identities were merged. When performing the statistical evaluation of the results, we found that this approach misses aliases where the person's real names and addresses are substantially different, e.g., ‘‘`John Smith <john@hotmail.com>`’’ and ‘‘`J.S. <jsmith@freemail.com>`’’. However, these cases do not occur frequently and could be resolved manually.

Overall, we find an average of 10% of email addresses to be aliases for the same person, underlining the importance of merging multiple aliases before studying mailing list data.

3.5 Related Work

The challenges presented in this work have many implications for applications of mailing list data mining in research. In the past, many of these challenges have been described only anecdotical or as side-notes.

Bird et al. mine mailing lists to study social networks [Bird2006a; Bird2006b]. They identify the multiple alias problem and propose the use of a clustering algorithm to merge identities.

Herraiz et al. identify that mining repositories of open-source projects is a challenging task and propose general approaches to mining these repositories [Herraiz2006; Robles2005]. The mlstats tool used for their studies on GNOME mailing lists, mines information from email headers.

Kolcz et al. use text-mining approaches to detect near-duplicate email messages for spam identification [Kolcz2004].

Carvalho et al. use machine learners to identify signatures and quotations in email messages [Carvalho2004]. While this method can achieve good results, it needs a manual training step and sufficiently clean training data to perform well.

Tang et al. propose methods for cleaning plain text email messages, in order to make them accessible for text-mining and information retrieval [Tang2005]. Their work focusses on text transformation for natural language processing.

3.6 Summary

Mailing lists contain valuable information for maintainers of long-lived software projects. In order to make this information accessible for subsequent analysis steps it needs to be processed first. Many mailing lists document multiple years of project development. However, the email technologies that produce this mailing list data have changed several times over the past decade. As such, mailing lists contain a conglomeration of messages from different revisions of the email format. Using off-the-shelf techniques to process this data naively yields many risks for the validity of the resulting information.

Yet, for many of the presented issues no perfect, automated solutions exist. Email messages are substantially different from the much cleaner text sources used in related research areas like information retrieval. As such many of the text cleaning techniques used in text-mining and information retrieval cannot be readily applied to email communication. Hence, we see an opportunity for future work to refine mailing list data processing techniques.

3.6.1 Relevancy to this Thesis

Based on the challenges and solutions presented in this chapter, we have implemented a tool for mining communication data from email repositories. This tool, called `Mailbox Miner`, is publicly available under an open-source software license and can be downloaded from:

> `https://github.com/nicbet/MailboxMiner`.

This tool has already found wide adoption in the research area, e.g., [Jiang2013; German2013b; Thomas2011]. The Mailbox Miner tool forms the basis for our data extraction in later chapters of this thesis (Part III). In particular, we use the tool in Chapter 6 to mine developer discussions and investigate the relationships between different social aspects of these discussions to software

quality.

We use the tool in Chapter 7, to reliably mine code contributions and discussions on these contributions in the Linux Kernel project.

4

Mining Technical Information from Communication Data

In the previous chapter, we discussed the intricacies of mining communication data from email repositories. However, communication data mined from email, chat, or issue report comments, frequently consists of unstructured data, i.e., natural language text, mixed with technical information such as project-specific jargon, abbreviations, source code patches, stack traces and identifiers. Technical artifacts represent a valuable source of knowledge about the software. The intertwining between natural language and technical content make the separation of these two types of text challenging. In this chapter, we present a general-purpose, yet lightweight approach to extracting technical information from unstructured data. Our approach is based on existing spell checking tools, which are well-understood, fast, readily available across platforms and impartial to different kinds of technical artifacts. Through a handcrafted benchmark, we demonstrate that our approach is able to successfully uncover a wide range of technical information in unstructured data, and provides a statistically significant improvement over the state-of-the-art (+23% precision, +16% recall).

4.1 Introduction

Every software system has a unique history of design decisions, software changes, as well as development and maintenance effort. This history is captured throughout the development process in the variety of repositories used to store data during the collaborative development process. As this data contains the knowledge and rationale behind the evolution of a software system, it is valuable for many different fields, in particular program comprehension, and hence should be made available to practitioners and researchers alike.

However, much of the information surrounding the development process comes in the form of *unstructured data* [Bettenburg2010b], which is conceptually different from the sources of structured data that researchers have used in previous research. Structured data (e.g., source code) is well-defined and can be readily parsed and understood by computer machinery. Unstructured data (e.g., developer communication, issue reports, documentation, email or meeting notes [Shihab2009b]), consists of a mixture of natural language text and *technical information*, such as code fragments, abbreviations, references to objects in the source code, file names, logging information or project-specific terms. As such, mining unstructured data is challenging: it is meant for the exchange of information between humans, rather than automated processing using computer machinery. Figure 4.1 presents an example of technical information commonly found in unstructured data.

Recent approaches for discovering technical information in unstructured data [Bacchelli2010b; Bettenburg2008b] have focussed on recognizing and extracting only particular types of technical information, such as class names [Bacchelli2010b], stack traces, or patches [Bettenburg2008b]. In order to resolve the inherent ambiguities between natural language text and technical information, these approaches are highly specialized and tailored towards their specific use cases, and limited in their scope. Furthermore, many kinds of technical information (e.g, project-specific jargon or abbreviations) cannot be extracted by any of the existing techniques.

As a first step towards a lightweight, general-purpose approach to uncovering technical

Build ID: M20070212-1330

Steps To Reproduce:
1. Create a plugin for eclipse that includes a key binding for "M1+S" (ie. Alt+S)
   where S is any letter that is used as a mnemonic in one of the top level
   menus. Since eclipse uses "S" as the mnemonic for Help > &Software Updates,
   "S" is sufficient.
2. Launch the plugin as part of Eclipse IDE
3. Press Alt+H to bring down the Help menu (to go along with our example in #1)
   BUG: Notice "Software Updates" is missing its mnemonic.

More information:
The code after "if (callback.isAcceleratorInUse(SWT.ALT | character))" inside
Eclipse's MenuManager.java removes the mnemonic, but it seems like Eclipse
should be checking "isAcceleratorInUse" only for top level menumanagers like
File,Edit,...,Help, etc. :

```
 /* (non-Javadoc)
   * @see org.eclipse.jface.action.IContributionItem#update(java.lang.String)
   */
public void update(String property) {
IContributionItem items[] = getItems();

for (int i = 0; i < items.length; i++) {
items[i].update(property);
}
[...]
}
```

Any status on this bug?

I'd consider any contributions for M6 (API) or M7 (non-API) [...]

A 3.5 fix would be to make that behaviour optional in MenuManager with API and
off by default early in 3.5, and to have the WorkbenchActionBuilder contributed
MenuManagers and actionSets/editorActions contributed MenuManagers turn it on
(if I can find MenuManagers in the correct place).

I'd like us to work with the SWT team to make sure we understand what the
correct platform behavior is, and make sure that we aren't getting in the way
of that. The current behavior (i.e. turning off mnemonics) seems odd to me, in
general. If we're going to fix this, we should fix it properly.

Figure 4.1: Examples of technical information uncovered by a prototype implementation of the approach proposed in this chapter. (Eclipse Platform Bug #208626).

information in unstructured data, this chapter presents an approach that makes use of state-of-the-art tools for checking and correcting the spelling and grammar of electronically written texts. Technical information is conceptually different from natural language text: it often consists of words that are not part of standard language dictionaries, violate grammatical conventions, and do not respect morphological language rules. These characteristics render modern spellcheckers ideal candidates for lightweight classifiers of natural language.

Through a case study on unstructured data from mailing list and issue report repositories of two open-source projects, we demonstrate the capability of our approach to uncover technical information inside unstructured data, while at the same time being resistant to reporting actual spelling or grammar mistakes.

4.1.1 *Contributions*

The work presented in this chapter makes the following contributions to the research area: first, we present a *novel, lightweight approach for separating technical information from natural language text*. Second, we present an empirical evaluation of this approach based on *a benchmark suite* created through manual annotation and analysis of real-world unstructured data.

4.1.2 *Organization of this Chapter*

The rest of this chapter is organized as follows. Section 2 presents an overview of related work and background. Section 3 presents our approach from both a conceptual and an actual implementation perspective. In Section 4, we present the evaluation of our approach through a hand crafted benchmark on developer email and issue report discussions. We conclude our work and present future research opportunities in Section 5.

4.2 Background and Related Work

Past research has been concerned with the extraction of technical information from software repositories, to assist program comprehension [Malik2008; Nurvitadhi2003], understand historical changes [Zimmermann2004; Mockus2000b; Hassan2004] and predict future changes [Zimmermann2005], and to measure and analyze different dimensions of historic software development to help practitioners make informed decisions in the future, and predict software errors [Zeller2008; Sliwerski2005; Nagappan2006b].

Fischer et al. and Sliwerski et al. were among the first to use technical information (issue report identifiers) embedded in the natural language text descriptions of changes in commit messages, to link software changes to defects [Fischer2003a; Sliwerski2005].

Mockus et al. show that the text describing a change recorded through commit messages is essential for understanding the rationale behind changes and emphasizes the importance of natural language documentation for practitioners and researchers alike.

Recent research concerned with information in unstructured data has mostly focussed on establishing traceability links [Antoniol2002; Marcus2003; Bacchelli2010b] between source code and documentation surrounding the development process, summarizing communication [Shihab2009b; Hindle2009], and bug triage [Anvik2006].

The most closely related work to this paper is the work on techniques to uncover source code entities in e-mails [Bacchelli2010a], and classifying text into source code and natural language text on a line-level granularity [Bacchelli2010b]. Bettenburg et al. presented the use of island parsing and specialized heuristics based on regular expressions to extract structural information from bug reports [Bettenburg2008b].

Our work is different from past research in the area, in that we aim to uncover technical information in unstructured data by using spell checkers as a lightweight classification-proxy to determine which parts of the text are natural language text and which parts are not. Our approach aims at being general enough to be readily available for any kind of input beyond commit messages, bug reports or e-mail. Furthermore, our approach does not focus on a particular type of technical information, such as bug report identifiers or source code entities, but rather to return information in unstructured data that is not considered natural language text. Such a general set of technical information has the advantage that it can be easily pruned later on, by applying further heuristics to retain only a particular kind of technical information of interest.

4.3 Approach

In the following, we present our approach from both a conceptual perspective and the concrete perspective of our working prototype.

4.3.1 Conceptual Approach

For the technique presented in this chapter, we use existing spellchecking tools to untangle natural language text and technical information from unstructured data. Many of today's state of the art techniques for spellchecking use morphological language analysis, which describes the identification and description of the smallest linguistic units that carry a semantic meaning, called *morphemes*. As such, morphemes are different from the concept of a single word: one or more morphemes composed form a word. For example, the English word "unbearable", is composed of three morphemes, "un", "bear", and "able". This kind of analysis is able to effectively cope with compound words, inflection and other peculiarities of natural language,

while at the same time being sensitive to text (technical information) that does not adhere to the morphological rules.

For the purpose of our study, we define technical information as those parts of unstructured data that is not natural language text. This definition includes, but is not restricted to: source code, file names, technical terms, project-specific jargon, source code entities (such as classes or identifier names), or abbreviations.

4.3.2 *Concrete Approach*

In order to uncover technical information, we first transform the input text in a stream of tokens by splitting the input text whenever we encounter one or more whitespace characters, or punctation followed by a whitespace (sentence delimiters). This is a common approach for morphological language analysis of Western text, where words are delimited by whitespace. If we were to apply our method to Chinese or Japanese input text, we would need to modify tokenization accordingly.

After thorough testing of 15 open-source spellchecking tools, we select the following three popular tools for further study. `Hunspell` is an open-source spellchecking and morphological language analysis framework, which has found extensive use in the `OpenOffice` and `Mozilla` application suites. `Jazzy` is based on the *double metaphone* phonetic language analysis algorithm [Philips2000], which transforms words into phoneme codes and compares these to a user-defined dictionary. `JOrtho` performs spell checking by comparing a given input word to large word dictionaries compiled from the `Wiktionary`[1] project.

Next, we run the spellchecker on each token and flag it, depending on wether the spellchecker reported a spelling error or not. Since our goal is to find technical text, rather than spelling mistakes, we iterated over each flagged token in a second pass, executing three different, simple heuristics. If at least one heuristic holds on a flagged token, we mark the token as belonging to

[1] http://wiktionary.org

```
(\b([A-Z_][a-z_0-9]*)+[A-Z_0-9][a-z_0-9]+
                    ([A-Z_0-9][a-z_0-9]*)*\b)
(\b([a-z_][a-z_0-9]*)+([A-Z_0-9][a-z_0-9]*)+\b)
(\b([A-Z_][A-Z_0-9]+)\b)
(\b(([A-Z_][a-z_0-9]*)+([A-Z_0-9]+))\b)
```

Figure 4.2: Regular Expressions Used to Identify Camel Case.

the domain of technical information. The heuristics we use are described in the following.

H1:Camel Case

We consider the following four cases of camel case to be indicators of technical text: (1) the standard case `CamelCase` is often used for type names and references to source code entities; (2) the interior case `camelCase` is often used for identifier names; (3) capital letters at the end `CamelCASE`, and (4) all capital letters `CAMELCASE`, are often used in abbreviations. We implemented this heuristic with a simple pattern matching using the regular expressions presented in Figure 4.2.

H2:Programming Language Keywords

We compiled a comprehensive list of reserved keywords for the JAVA, C, C++, C#, Pascal, Delphi, Perl, PHP, Bash, HTML and JavaScript languages from the official documentation of these programming languages. If a token is flagged as a spelling mistake, but matches one of the keywords in this list, it is highly likely to be part of a source code fragment, and we treat the corresponding part as technical information of type “code”.

H3:Special Characters

Natural language words usually do not contain special characters within their word boundaries. When a token is flagged as a spelling mistake, we count the number of non-alphanumeric

characters in the token and consider it as technical text, if we find more than two special characters.

4.4 Evaluation

We evaluate the ability of each of the three selected spellchecking frameworks to untangle natural language text and technical information from unstructured data through a hand-crafted benchmark. We performed a random sampling of 20 issue reports from the `ECLIPSE` project and 20 email discussions from the `PostgreSQL` developer mailing list, containing source code, stack traces, patches and other technical entities. The size of this random sample describes our results across the overall population at a confidence interval of 15%. We annotated technical information in each document by hand, using a graphical tool written for this purpose.

The tool allows the user to select a portion of the text inside a text viewer and annotate it as technical text. When the user then activates a particular spellchecking framework, the portion of the text will be annotated with different colors depending on whether the spellchecker flagged a portion of the text which was previously not annotated (false positives, FP), which was annotated but not flagged by the spellchecker (false negatives, FN), and which was flagged, as well as previously annotated (true positives, TP).

We then measure the average *precision* and *recall* of each spellchecker S_i across all documents in the benchmark. These measures are defined as:

$$\begin{aligned} Precision(S_i) &= \frac{|TP_{S_i}|}{|TP_{S_i}+FP_{S_i}|} \\ Recall(S_i) &= \frac{|TP_{S_i}|}{|TP_{S_i}+FN_{S_i}|} \end{aligned}$$

The results of our manual benchmark are presented in Table 4.1. Overall, we found all three spellcheckers to perform well, with a precision between 84.16% and 88.01%, and a recall

Tool	Precision	Recall
JOrtho	88.01%	64.31%
Jazzy	84.16%	68.30%
Hunspell	86.40%	68.34%

Table 4.1: Results of Benchmark

between 64.31% and 68.34%. The most common error across all tested spellcheckers with respect to precision were spelling mistakes that were not distinguishable from technical text, such as *"found.We"*, or *"another(no one else would)"* resembling package names or method calls. The rather moderate recall can be mainly attributed to the resemblance of many technical items to natural language text, for example source code identifiers are often words present in English dictionaries, e.g., *"Task"*.

In addition to our fine-grained performance analysis, we conducted an experiment to compare the use of technical information uncovered by our approach for a more specialized task, presented by Bacchelli et al., i.e., extracting source code from email [Bacchelli2010b]. As the latter approach operates on a line-level granularity, we augmented our spellchecking-based technique to consider a line of text as source-code, if more than seventy percent of text in that line was flagged as technical information by our approach.

We applied both approaches to the same data set of 40 documents (20 issue reports and 20 developer emails) used in our previous evaluation. The baseline for this experiment was established through the same benchmark annotation tool used in our previous evaluation. In terms of precision, our approach was able to classify 89.27% of lines correctly as source code, compared to 66.13% percent of lines correctly classified by the state-of-the-art technique. In terms of recall, our approach was able to recognize 86.46% of all source code lines correctly, compared to 69.37% of source code lines recognized by the state-of-the-art technique. All performance differences are statistically significant at $p < 0.001$.

Overall, the lightweight approach to extracting technical information from communication data, presented in this chapter improves on the best existing techniques by 23.14% (precision) and 16.09% (recall) respectively.

4.5 Conclusions and Future Work

In this chapter, we presented a lightweight approach to finding technical information in unstructured data, as a first step to making technical information readily available for researchers and practitioners. The evaluation of our approach demonstrates that readily available spellchecking tools, when paired with additional lightweight heuristics, are able to successfully untangle technical information and natural language text. In future work, we plan to study the use of additional heuristics to increase recall and carry out a more detailed evaluation on different kinds of technical information through an extended benchmark.

4.5.1 Relevancy to this Thesis

The work presented in this chapter forms the basis for the data analysis presented in Chapter 6. In particular, we use the techniques presented in this chapter in an extended version of our `infoZilla` tool, to extract technical artifacts from developer discussions. We then use this information to study the relationships between discussions that reference technical artifacts and software quality. Furthermore, we use the techniques presented in this chapter for the mining of code contributions in Chapter 7 from the Linux Kernel project. There, the objective is to reliably detect and extract code contributions that are embedded in natural language text in the form of patches.

5

Linking Communication Data to Source Code

When discussing software, practitioners often reference parts of the project's source code. In the previous chapter, we have presented a lightweight approach to separating technical information such as class names, function names, stack trace, or source code examples from natural language text. Such references have different motivations, such as mentoring and guiding less experienced developers, pointing out code that needs changes, or proposing possible strategies for the implementation of future changes. Knowing which code is being talked about the most can not only help practitioners to guide important software engineering and maintenance activities, but also act as a high-level documentation of development activities for managers. In this chapter, we present an approach based on clone-detection as specific instance of a code search based approach for establishing links between code fragments that are discussed by developers and the actual source code of a project. Through a case study on the Eclipse project, we explore the traceability links established through this approach, both quantitatively and qualitatively, and compare our approach to classical linking approaches, in particular change log analysis and information retrieval. The results of our study show that the links established through fuzzy code search are conceptually different than traditional approaches based on change log analysis or information retrieval.

5.1 Introduction

In "The Cathedral and the Bazaar" [Raymond1999], Eric Raymond notes that one of the main advantages of open-source development is the reduced rate of software defects grounded in Linus' Law, i.e., "a direct result of the increased communication among developers about the source code". Understanding the impact of such developer communication on software quality has been the focus of recent research [Bacchelli2010a; Bettenburg2010a] and is based on the explicit and implicit knowledge of developers that is recorded during the development of a software system.

Implicit developer knowledge is embedded in a variety of repositories, such as mailing list archives, modification requests, issue reports, the source code itself and accompanying documentation. Often, this implicit knowledge is of informal nature and consists of a mixture of natural language texts and structural elements that refer to the project's source code. Links between the source code and the surrounding documentation and communication have been recognized in the past as an important factor for effective software development, and as a result, software engineering research spends much effort in uncovering such traceability links [Oliveto2007].

Past approaches to uncovering traceability links between documentation and source code are commonly based on information retrieval [Antoniol2002; Marcus2003; Lucia2007; Jiang2008], natural language processing [Maarek1991; Antoniol2000] and lightweight textual analyses [Fischer2003a; Bacchelli2009]. Each approach, however, is tailored towards a specific set of goals and use cases. For example, when linking code changes to issue reports by analyzing transaction logs [Sliwerski2005], we observe only the associations between a bug report and the final locations of the bug fix, but miss the bug fixing history: all the locations that a developer had to investigate and understand before he could find an appropriate way to fix the error.

In this chapter, we present, evaluate and compare different approaches to finding traceability links between the communication surrounding **M**odification **R**equests (MRs) and the source

code of a software project. For this purpose, we propose a new approach that uses token-based clone detection as an implementation of fuzzy code search for discovering links between code fragments mentioned in project discussions and the location of these fragments in the source code body of a software system. In a case study on the ECLIPSE project, we first extract source code fragments from bug report discussions and then use the CCFinder clone detection tool, a readily available implementation of fuzzy code search, to identify all occurrences of the extracted code fragments in the software system's source code. We explore the value of the resulting traceability links through a quantitative evaluation and compare the resulting traceability links to those established by two classical approaches: change log analysis, which is the state-of-the-art for linking issue reports to source code, and information retrieval.

5.1.1 Contributions

Our work makes the following contributions to the research area: first, we *establish a new class of traceability links* that link code fragments contained in project discussions to the actual occurrences of these fragments in the source code body of a software system. Second, we report on *a qualitative and quantitative analysis* of our approach. Third, we *demonstrate an example application* of this new class of traceability links through identification and visualization of those parts of the software system that are discussed the most.

In the future, we envision traceability links established by our approach to be used to assist practitioners when browsing issue reports. A sample application would be an enhanced BugZilla system as illustrated in Figure 5.1, which assists the bug fixing process by identifying code fragments contained in the corresponding issue report discussion and points developers to the locations in the project's source code that contain similar or identical code.

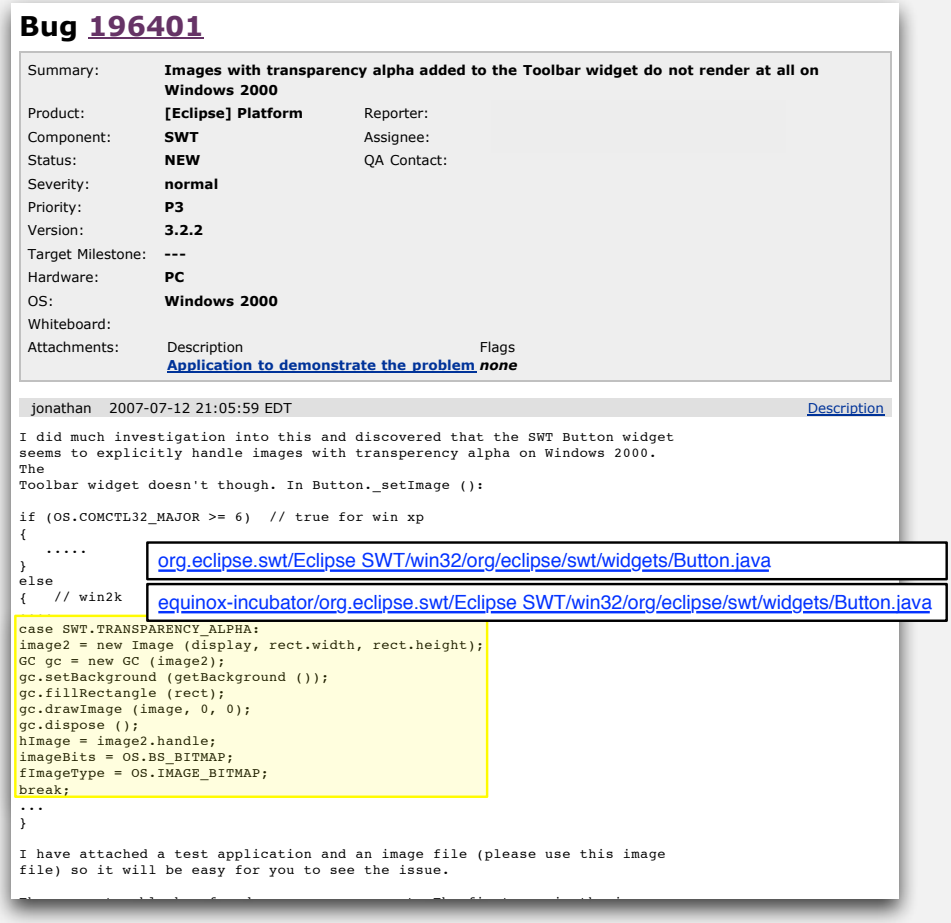


Figure 5.1: Example of an enhanced bug report system that points developers to similar code.

5.1.2 *Organization of this Chapter*

The rest of this chapter is organized as follows: in Section 2, we present related work to our study, followed by a discussion of the limitation of existing approaches in the context of linking code fragments contained project discussions to source code. Together with this discussion, we present our fuzzy code search-based approach to traceability linking in Section 3. In Section 4 we present the design and results of a case study on the ECLIPSE software system, followed by a sample application of the obtained traceability links to uncover the parts of the software that

are discussed the most (Section 5). We conclude our work in Section 6 with a discussion of our results and future research avenues.

5.2 Background and Related Work

Previous work in the area of traceability link recovery between source code and documentation can be categorized into three groups: information retrieval-based approaches, approaches that analyze the change logs of transactions to the version control system, and lightweight textual approaches that scan documents for code entities. In the following we summarize the research in each of these categories.

5.2.1 Information Retrieval Approaches

Frakes and Nejmeh pioneered the use of information retrieval approaches to support software reuse. Their CATALOG system implemented an interactive search engine for source code documents based on search terms supplied by the user [Frakes1987].

Maarek et al. extended this idea in their work on automatically constructing software libraries with the help of information retrieval methods [Maarek1991]. Attributing the moderate success of software reuse as a lack of a central library for locating and understanding code and documentation, the authors propose to use information retrieval methods to group sets of unorganized documents into software libraries, thus connecting source code with surrounding documentation.

Antoniol et al. recognized that the domain-specific knowledge of developers is implicitly encoded in such surrounding documentation in the form of mnemonics for identifiers that capture high-level program concepts [Antoniol2000].

In an extension to their former work, Antoniol et al. studied the use of vector-space information retrieval models for recovering traceability links between source code and free-text documentation [Antoniol2002]. In their case studies on two software systems, Antoniol et al. found that both approaches yield very high recall, with both approaches finding up to 100% of the existing links.

Marcus et al. extend Antoniol's idea by investigating the use of a novel vector-space information retrieval model for traceability link recovery [Marcus2003]. In their work, they demonstrate the suitability of Latent Semantic Indexing (LSI) to the domain of source code and documentation, while being computationally less expensive than the previous approaches by Antoniol et al. [Antoniol2002].

Cubranic et al. link source code to documents, tasks, persons and messages in their HIPIKAT project memory system [Cubranic2005]. For a given artifact, they establish links to similar artifacts by calculating document similarities based on an LSI vector space model similar to Marcus et al. [Marcus2003]. Through two case studies they show that the approach successfully provides pointers to files needed for the specific modification tasks.

In a study on automatic generation of traceability links between arbitrary software artifacts, De Lucia et al. extend a software artifact management system called ADAMS with a traceability recovery approach based on LSI [Lucia2007]. Their case study revealed that information retrieval-based traceability link recovery suffers from a high number of false positives requiring much manual effort from users to discard incorrect links.

Information retrieval techniques, especially vector-space models have been demonstrated to be successful for automatically identifying semantic connections between source code and surrounding documentation. Jiang et al. were the first to note that previous techniques could not effectively and automatically deal with software evolution [Jiang2008]. As a solution to the problem of changing documents over time, they proposed an incremental LSI technique and implemented the approach in an automated traceability link evolution management tool.

Asuncion et al. presented another incremental IR approach, which uses latent Dirichlet

Technique	**Intended Purpose**
Information Retrieval [Antoniol2002; Marcus2003; Asuncion2010]	Establishing traceability links for software requirements.
Change Log Analysis [Fischer2003a; Sliwerski2005; Bird2009b]	Associating changes to the source code with issue reports.
Lightweight Analysis [Bacchelli2009; Bacchelli2010c]	Linking mailing list discussions to source code entities mentioned in the discussion.
Code Search (This work)	*Linking code fragments extracted from project discussions to their location in the project's source code.*

Table 5.1: Overview of existing linking approaches.

allocation (LDA) for capturing traceability links [Asuncion2010]. They show in a case study that LDA performs as well or better than LSI with respect to precision and recall of the captured traceability links, while being computationally less expensive than LSI.

5.2.2 *Change Log Analysis*

In addition to information retrieval approaches, there exist a variety of specialized approaches for establishing traceability links. These approaches rely on implicit knowledge about software development repositories to establish traceability links between source code and documentation.

Two of the most prominent approaches, which link source code to bug reports, were introduced by Fischer et al. [Fischer2003a; Fischer2003b] and Cubranic et al. [Cubranic2005], who discovered that developers tend to include bug report identifiers in the change logs of version control system transactions. This information can be used to link the files affected by a transaction to the bug report mentioned in the change log message.

Sliwerski et al. [Sliwerski2005] refined this approach and introduced a set of heuristics that measure the syntactic and semantic relevance of links to cut down the number of false positive links reported by this method.

Associating source code and issue reports through the analysis of change log messages has been widely adopted in the defect prediction community, forming datasets that are at the core

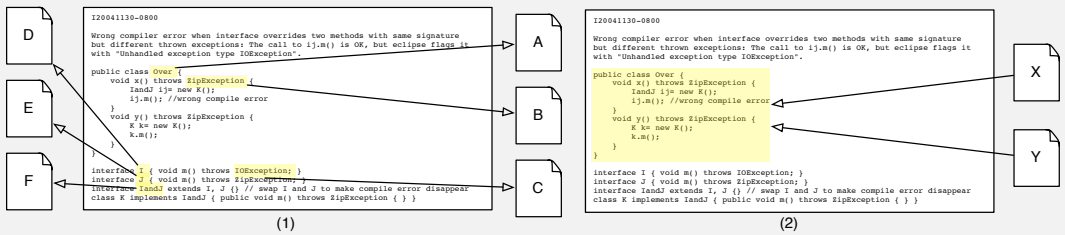


Figure 5.2: Lightweight textual analysis finds traceability links from code entity names to files implementing those entities (1). Fuzzy code search links the entire code fragment to the actual occurrences of the fragment in the projects' source code (2).

of many research efforts in the area [Kim2006; Schroter2006; Moser2008].

However, Bird et al. note that these datasets suffer from inconsistent linking and represent only a smaller sample of links from the population of all possible links between issue reports and the source code [Bird2009b].

5.2.3 *Lightweight Textual Analysis*

As information on bug report identifiers is limited to commit log messages, change log analysis cannot be used to link other forms of documentation to source code artifacts. Bacchelli et al. hence proposed a set of lightweight techniques [Bacchelli2009] to establish traceability links between mailing list discussions and source code, based on the recognition of entity names, such as classes definitions or method names, inside the textual contents of messages and linking them to their corresponding implementation files.

5.2.4 *Limitations of existing approaches*

Table 5.1 summarizes existing approaches discussed so far. Each of these approaches has been designed to meet a specific goal and they have been shown to perform well for their intended use cases. In this paper, we want to consider the context of establishing traceability links between

issue reports and source code files. In particular, we focus on establishing links between code fragments contained in bug report discussions to their occurrences in the project's source code. Within this intended use-case we identified the following limitations for the applicability of existing approaches:

Information retrieval (IR) methods, such as Latent Semantic Indexing (LSI), latent Dirichlet allocation (LDA), or Vector-Space Models (VSM) are designed to identify commonly occurring concepts and patterns across combined collections of source code documents and surrounding documentation. However, IR techniques rely on a user-defined number of dimensions that limits the amount of concepts derived and thus the specificity of uncovered traceability links. Additionally, IR techniques usually rely on a repetition of text features in order for them to emerge as concepts, whereas code fragments are often small (compared to the size of whole files of source code), and often lack the required repetition of features that is required in this approach.

Change log analysis based traceability linking requires that an issue has been filed through the bug report system, and that the issue has to have led to an actual change of the source code. Additionally, when fixing bugs, developers commonly discuss different implementations and often discuss many parts of the source code with the final fix possibly taking place in a completely different part of the source code. Links established through change log analysis record the final location of a much more involved and complex process and ignore the history of the bug fixing process.

Lightweight textual analysis approaches, for example as presented by Bacchelli et al. [Bacchelli2009], focus on linking names of source code entities mentioned in developer discussions, such as identifiers and types, to the implementation files of these entities. These links are conceptually different from our approach, as we do not want to link entity names to the implementation of these entities, but whole fragments of source code examples to the locations

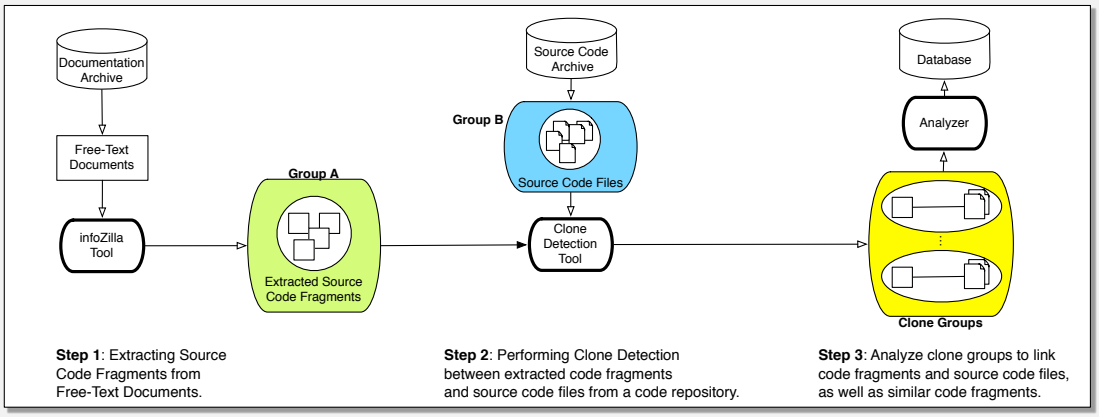


Figure 5.3: Overview of a clone-detection based approach to locate code fragments in a project's code base.

in a projects source code where these fragments occur. To illustrate the conceptual difference of the links created by this approach, consider the bug report shown in Figure 5.2. In this example, lightweight textual analysis would recognize the types `Over`, `ZipException`, `IOException`, and interface types `I`, `J`, and `IANDJ`. For each recognized type, a traceability link is established to those source code files `A` to `F` that define and implement these types (e.g., `some/path/ZipException.java`).

In fuzzy code search-based traceability linking, the complete code fragment contained in the discussion would be recognized as a smaller subset of code that is contained in some source code files `X` and `Y` (e.g., `some/path/TestCase1.java`).

5.3 *Code Search Based Traceability Linking*

In order to locate source code fragments contained in project discussions within the source code body of a software system, we propose to use clone-detection as a readily available fuzzy code search implementation. An general overview of our proposed approach is illustrated in Figure 5.3.

Step 1: First, we use the `infoZilla` tool [Bettenburg2008b] to extract source code from documents obtained from a documentation archive (Step 1). The `infoZilla` tool uses a combination of regular expressions and island parsing [Moonen2001] to identify and extract source code regions from free-form text documents with high reliability. At first, the complete textual input is treated as "water". Using regular expressions it identifies common program elements, such as assignments, method calls or loops. These elements form "islands" in the water. The tool then examines the text surrounding each island to determine whether this text is code and grows the island accordingly. For a more detailed discussion of the tool we refer to our previous work [Bettenburg2008a]. Each extracted code region is stored in a separate file, uniquely identifying the original discussion document from which the code was extracted. We refer to the complete collection of extracted code regions as "Group A".

Step 2: In order to discover where each source code fragment of Group A appears in the source code of the project (we refer to the collection of source code files of a project as "Group B"), we use a token-based clone detection tool to carry out a fuzzy textual search. We call this *code search*. We stress that the fuzzy search aspect is critical: during the manual inspections of source code and code fragments carried out during design and refinement phase of our work, we found that in practice, occurrences of code fragments in the actual source code body of the project are often slightly modified versions (e.g., by normal evolution of the source code, or to adapt code examples to particular APIs) of the original fragment contained within discussions.

Step 3: Clone detection tools usually report their findings in terms of clone pairs and clone groups. A clone pair associates a source code region in a file f_1 with a corresponding duplicated code region in another file f_2. All clone pairs with identical duplicated code regions are grouped together and form a clone group. We analyze all clone groups for clone pairs that associate files with extracted code fragments (Group A) to source code files (Group B). A traceability link as established by our proposed approach is hence a tuple containing a unique clone group identifier, a file path that corresponds to a code fragment, a file path that corresponds to a source code

file, and a description of the exact location of where the code fragment occurs within the source code file. We store all traceability links in a database for further analysis.

5.4 Case Study

In this section, we present a case study on the ECLIPSE software system. We apply our proposed approach to discover traceability links between discussions attached to issue reports contained in the projects BugZilla bug tracking system, and the complete source code of the software contained in the project's CVS software archive.

We first perform a quantitative evaluation that illustrates the performance of our proposed approach with respect to the amount and validity of the established traceability links. We then proceed with a qualitative analysis that first compares traceability links generated through fuzzy code search to traceability links generated through change log analysis, the state of the art method to link issue reports to source code files contained in a version control system. We finish our analysis with a discussion of the use of information retrieval based traceability linking and its shortcomings in the presented use case.

5.4.1 Data Collection

Based on past interest in establishing links between developer discussions and code [Bacchelli2010c], as well as the significance of links between issue reports and source code for defect prediction [Bird2009b], we chose to use the descriptions and discussions attached to issue reports of ECLIPSE as our main source of project discussions. We followed the approach by Zimmermann et al. [Zimmermann2008b], and extracted a total of 211,843 issues from the BugZilla issue tracking system of the ECLIPSE project, that were filed from ECLIPSE version

2.0 until ECLIPSE version 3.2. Additionally, we obtained a snapshot of the complete software archive of the ECLIPSE 3.2 release. This snapshot contains both, the project's source code, and a complete record on the history of all changes to the source code that were carried out by developers.

In order to perform clone detection, we use one of the most popular token-based clone detection tools, CCFinder [Kamiya2002]. This choice is mainly motivated by the high recall of token-based clone-detection, paired with its good scalability. Furthermore, token-based clone detection approaches have the advantage over other approaches that they can work with uncompilable code, such as commonly found in code fragments of discussions.

We adapted the CCFinder tool, which was originally written for the Windows platform, to a 64 bit version of Ubuntu Linux Server 9.10. These modifications allowed us to perform our experiments in main memory, greatly increasing performance and allowing us to work on the complete copy of the project's source code. Using the CCFinder tool, we executed an inter-group clone-detection between the code extracted from issue reports (Group A in Figure 5.3) and the project's source code from the version control system (Group B in Figure 5.3). Code clones are reported by CCFinder as a set of clone groups.

Each clone group is associated with a unique identification number and contains a sequence of clone pairs that describe the exact locations of a code clone between two files f_1 and f_2. As we ran the clone detection tool with the option to carry out an inter-group analysis, each clone pair reports on the occurrence of a cloned instance of code from a file belonging to group A in a file from group B.

Unfortunately, CCFinder reports one pair for each possible permutation of pairs that can be obtained from the set of clones in a clone group. In order to prune this representation, we transform the results reported by CCFinder in the analysis step and identify unique triples of clone group identifier, absolute filename, and the exact cloned code. These triples are then stored in a database for further analysis.

Using the approaches described in Chapter 3 and Chapter 4, we extended our original

infoZilla [Bettenburg2008b] tool. We use this extended tool to extract technical information in the form of source code fragments from the discussions attached to the 211,843 ECLIPSE issue reports. A total of 33,301 of these discussions contained source code fragments, and of these, a total of 17,748 discussions contain code fragments with a length of more than 30 source code tokens, which is the minimum amount of tokens needed for clone detection by CCFinder. From those 17,748 discussions, we removed another 10,042 instances that CCFinder failed to transform into a token stream. The transformation requires a certain amount of context, such as basic blocks, for inferring token types, and code fragments in these instances did not include enough context for CCFinder's transformation. This leaves us with a total of 5,511 developer discussions that form the base of our analysis.

5.4.2 *Quantitative Analysis*

Overall, our approach was able to establish a total of 47,783 traceability links, which connect 3,865 out of 5,511 (70.13%) of the discussions attached to ECLIPSE issue reports to a total of 13,581 out of 51,600 (26.32%) ECLIPSE source code files. We found that on average, each discussion (which might contain multiple different code fragments) was linked to 5.67 source code files.

Our approach failed to correctly process 1,646 of the 5,511 (29,87%) discussions. In order to better understand, why these discussions could not be linked to the project's source code, we performed a manual investigation. Overall, we identified the following two main causes:

1. **Unrelated code.** Many discussions contain code fragments that are not part of the project's source code body. For example, bug #99986 describes a problem with ECLIPSE's handling of inheritance in a specific build. However, the code example used to demonstrate the problem was not (yet) part of the ECLIPSE 3.2 source code.

2. **Code evolution.** Source code, especially code discussed in issue reports, often undergoes significant changes during the evolution of a system. For our experiment, we used a single snapshot of the software system's source code (version 3.2) within which we search for occurrences of the extracted code fragments. However, code fragments extracted from discussions might have undergone significant changes, especially in the case of discussions that refer to much earlier versions of the source code, up to a point that even fuzzy code search failed to relate the extracted code fragments to any part of the project's source code. One possible solution to this problem would be to take an evolutionary approach and also consider past snapshots of the software system source code, which are closer to the date of the discussion from which code fragments were extracted. We leave the investigation of this solution to future work.

5.4.3 *Qualitative Analysis*

To perform a qualitative analysis of our fuzzy code search-based approach, we first compare the traceability links established by our approach to traceability links established by the most prominently used approach in defect prediction: change log analysis. For this purpose, we implemented a change log parser, closely following the algorithm proposed by Sliwerski et al. [Sliwerski2005], as well as taking the enhanced heuristics described by Bird et al. [Bird2009b] into account. We applied this parser to the complete change history of the ECLIPSE version control repository and recorded all associations between issue reports and source code files.

Overall, the change log-based linking approach was able to link 16,722 issue reports to 23,079 source code files, with an average of 4.57 linked files per bug report. Of these 16,722 reports, a subset of 2,980 issue reports contain code fragments ($L \cap C$ highlighted in Figure 5.4a) and a subset of 507 reports were also linked by our approach ($O \cap C$ highlighted in Figure 5.4b). Taking the union of links established by both approaches, would result in a 20.01% increase in linked issue reports.

The very small intersection of the sets of issue reports linked by both approaches is notable: fuzzy code search-based linking creates many links between issue reports and source code that are not found by a change-log based approach and vice versa.

To explore the differences between both linking approaches, we randomly picked a sample of 10 reports each from the set of issue reports linked by both approaches ($O \cap L$), the set of issue reports linked by our approach but not by change log analysis ($O - L$), and the set of issue reports linked by change log analysis but not our approach ($L - O$). For each case in the random sample, we explore the established traceability links in the background of the corresponding issue reports and the discussion attached to them. We present our findings below.

1) Traceability links found by both approaches

The discussion of bug #31670, contains a sample class to illustrate a problem with the debugger. The source code used in the illustrative example creates a test case in the project's test suite. In addition to the original bug described, we learn that the corresponding transaction fixed another bug that was found during the fixing process. Traceability links of our approach point to the test case created for the original issue, whereas the traceability links of the commit log additionally point to the fix locations of both bugs.

In the discussion of bug #34593, a developer suggests a possible fix for the reported problem. Traceability links by both approaches point to the actual location of the fix that was carried out later on.

In the discussion of bug #51821, a developer proposes a sample patch to address the described issue. Based on the source code contained in the proposed fix, our approach links to the same location in the project's source code, that was later modified to fix the issue – the proposed patch was actually applied.

In the discussion of bug #61605, a developer proposes a new try-finally paradigm to enhance the robustness of a plugin. The proposed method is welcomed by peers and applied to plugins throughout the project. Traceability links of both approaches capture all modified files.

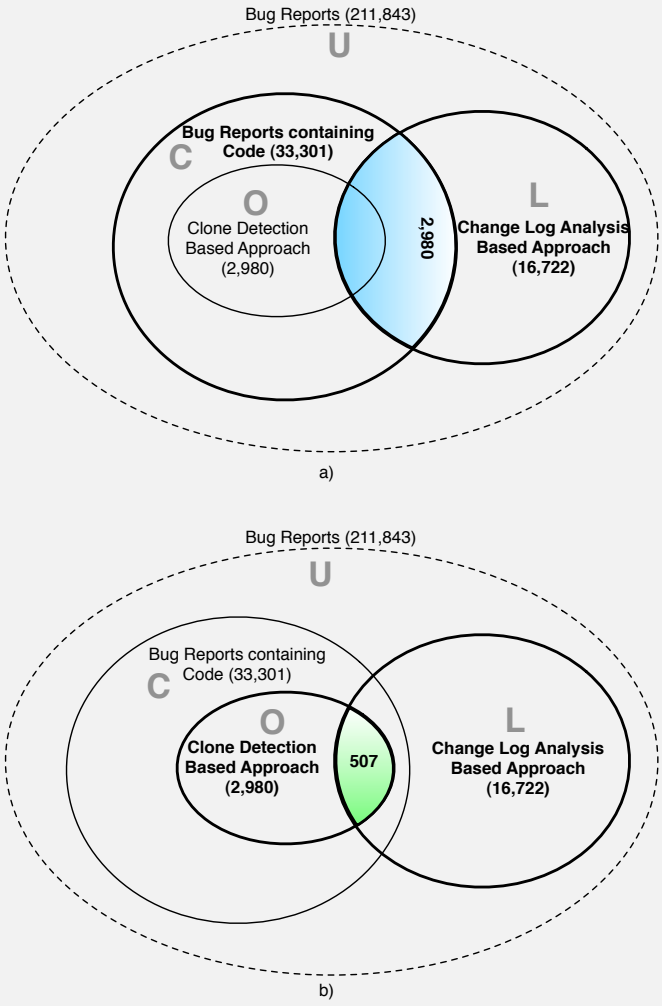


Figure 5.4: Venn diagrams of the set of issue reports linked through our clone detection based and a change log analysis based approach.

As reported in the discussion of bug report #87288, the reported issue and corresponding bug fix had a "large impact on downstream components". We can observe this impact through the traceability links established through our approach: 69 files contain code that is similar to the discussed code fragment. Of these, we observe 67 files that were changed in the bug fixing transaction.

During the course of fixing bug report #92017, developers added support for very large

images in main memory. In addition to a link to the fix location established through change log analysis, our approach establishes an additional link to the implementation of drag and drop support for images that uses the same approach as proposed in the bug fix.

In each case of bugs discussed in reports #93208, #93577, #105356 and #195763 the actual bug fix is applied in the source code locations linked through the discussed code examples. In all cases both approaches established the same traceability links.

In the case of bug #164939, a developers reports on a code fragment that he believes responsible for the issue. Based on the code fragment, our approach links to the location of the actual fix and an additional source code files that contains the same (erroneous) code, and which be believe should have been changed accordingly.

2) Traceability links found by code search, but not found by change log analysis

In the discussion of bug #21273, a developer provides an extract of the code he believes to be responsible for the filed issue. Traceability links established by our approach through this code fragment point to the actual location of a later fix. Change log analysis cannot establish links, as the transaction log does not contain references to any bug identifier.

In the discussion of report #45945, a developer reports a code example to illustrate the experienced problem. The bug report is later closed without a fix due to an operating platform incompatibility and no files are modified. Our approach however, establishes a link to a source code file containing code that is similar to the illustrative example, missing the correct link.

For report #62224, heuristics of the change log analysis based approach fail to identify the bug identifier. Our approach is able to establish a link to one of the two fix locations through the code fragment contained in the attached discussion.

In a similar way, change log analysis is not able to find the bug identifier in the change log of bug #65729. Our approach establishes links to all fix locations based on the code fragment mentioned in the bug report's discussion.

In the discussion of report #71047, a developer posts an example code to demonstrate that

he cannot reproduce the reported problem. The bug report is closed as WORKSFORME and no transactions take place. Traceability links established by our approach point to source code in the project that is similar to the developer's code example.

Through the code fragments discussed in reports #93467 #103266 and #106736, our approach establishes links to the exact fix locations. In all cases, change log analysis is unable to find these links, as no change logs ever contain the corresponding this bug identifiers.

Report #105447 is marked as a duplicate of report #137621 since a fix to #137621 reportedly fixes #105447 as a side effect. Even though both approaches find the same traceability links pointing to the fix location, they are associated with different issue reports.

Report #174125 is never fixed due to an operating system specific problem. As a result, no transactions take place that could be linked through change log analysis. However, the traceability links established by our approach point to source code that is similar to the presented code example.

We found that code extracted from issue reports #171912 and #171909 was linked to the same source code files and consequently grouped together in the results presented by the clone detection tool. Upon inspection of both issue reports we found that they are closely related and describe an issue that originates from the same parts of the project's source code.

3) Traceability links found by change log analysis, but not found by our approach

In the cases of reports #31573, #73908, #92579, #191862 and #202382, the code fragments contained in the discussions of these issue reports are used as illustrative examples. For example, the code in #92579 is intended to be visualized in the project outline view. In all cases, the code fragments are unrelated to the corresponding fixes and are not contained in the source code. Change log analysis is however able to establish traceability links to the final fix locations.

In the cases of reports #80455 and #182006, the code fragments contained in the discussions are below the minimum token length threshold and thus ignored by our approach. However, change log analysis is able to establish links to the fix locations of both bugs.

The code fragment extracted from the discussion of report #45468 consists exclusively of `javadoc`-style comments. `CCFinder` however, ignores code comments during clone detection and hence our approach cannot establish traceability links.

During the discussion of report #153932, a developer proposes code for a potential fix to the reported issue, but peer developers decide to modify a completely different part of the source code to address the problem. Our approach is unable to link the code of the proposed fix to any source code file.

5.4.4 *Using Information Retrieval for Traceability Linking*

In addition to comparing code search based traceability linking to change-log analysis, the state-of-the-art approach for linking issue reports to source code, we want to compare our proposed approach to traceability links obtained through information retrieval models. In particular, we use the Vector-Space Model (VSM) latent Dirichlet allocation, following similar approaches presented in the literature [Asuncion2010; Antoniol2002], as both approaches have been demonstrated valuable for establishing traceability links between source code and surrounding documentation.

A key finding in our previous work, and also very recently in related work, is that VSM is actually better than LDA for finding traceability links between source code and other related free-text documents [Rao2011]. VSM is more accurate (better precision), but LDA has better recall.

To perform traceability linking using LDA and VSM, we prepare our data following the standard approaches in the field: every extracted code fragment (Group A) and every source code file (Group B) is preprocessed by splitting identifiers, removing common English stopwords and stemming. We then use the combination of all documents in Group A and Group B to train an index. In the case of VSM, we weight each term by computing its term-frequency and inverse-document frequency (tf-idf).

The two IR models return a set of potential links between a given code fragment and source code documents, ordered by their similarity score (i.e., cosine distance for VSM and conditional probability [Wei2006] for LDA) in the model. In our case, each code fragment is potentially linked to hundreds or thousands of source code files, depending on if the two share common words or topics.

To determine the quality of the established links, we select a random sample for manual inspection. Sampling theory tells us that we need to inspect 64 samples to have a 10% margin of error and a 95% confidence interval. During our manual inspection, we inspected the top three links (by similarity score) for each code fragment. We found that, surprisingly, for each our 64 sampled code fragments, none of the top three links were accurate: the given code fragment was not found in the linked source code file. To illustrate why, consider the code fragment presented in Figure 5.5. The index models of both LDA and VSM would represent this code fragment by its four preprocessed terms `eobject`, `object`, `resourc`, and `content`. These terms are so general that the IR models will return thousands of possible matches, since thousands of source code files contain at least one of these terms.

```
EObject eObject = (EObject)resource
                  .getContents().get(0);
```

Figure 5.5: Code fragment extracted from ECLIPSE issue report #155726.

This poor performance is a consequence of several assumptions that IR models make. First, IR models are based on the "bag of words" model, meaning that the order of terms in each document is ignored. When searching for exact code fragments, this is an obvious disadvantage. Second, the preprocessing steps (especially splitting and stemming) used by IR models result in very general terms that are contained in many documents, as we saw in the example above. Third, the common similarity measures (e.g., cosine distance and conditional probability) are too general for our specific application, as they reward *any* shared words or topics, which is not

restrictive enough to eliminate false positive matches.

As a result, we conclude that IR based traceability linking is not suitable to reliably link code fragments contained in issue report discussions to occurrences of these fragments in the project's source code.

5.5 *Which parts of the software system are discussed the most?*

Previous research has noted the importance of traceability links for developers for software maintenance and source code comprehension [Antoniol2000], as they aid practitioners in creating mental models of the source code and providing hints to the location of specific code. We pick up this idea to demonstrate a sample application of the traceability links established by our proposed approach, as a natural extension to the association of code contained in project discussions to their occurrences in the project's source code.

Traceability links established through our approach are bi-directional: given a code fragment we can determine its location in the source code, but at the same same, given a location in the source code, we know in which discussion this source code was talked about. Through this association we can count the number of discussions that refer to a source code location.

Figure 5.6 presents a visualization of the most discussed components of the ECLIPSE software system. This visualization is inspired by Wattenbergs "Visualizing the Stock Market" [Wattenberg1999] and summarizes data from the beginning of the project until version 3.2. To increase visibility, we grouped source code locations at directory level. Every box represents a source code directory of the ECLIPSE software system. Boxes are coloured according to how much source code in each directory has been discussed in issue reports. Completely black boxes denote directories that contain source code that is discussed in less than 10 issue reports, and

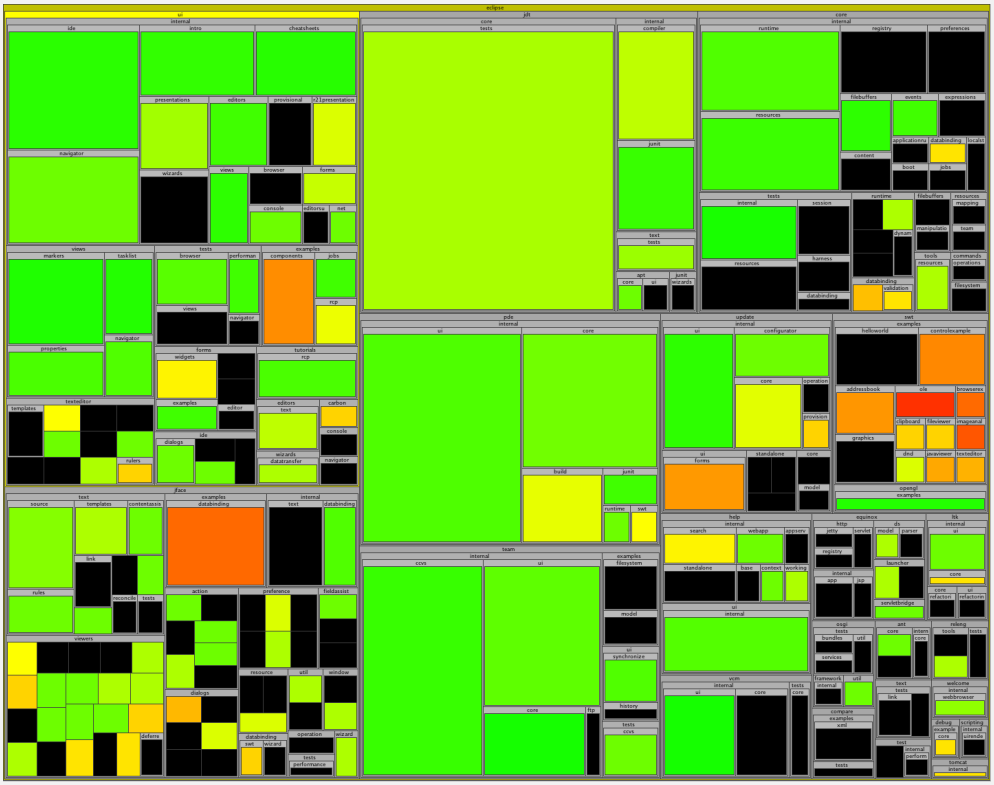


Figure 5.6: Visualization of the most-referenced source code in Eclipse issue reports. Brightness indicates the amount of references and ranges from dark/green (rarely referenced) to bright/red (referenced often).

the whiter a box, the more issue reports discuss the source code in this directory.

We can use this visualization to locate "discussion hotspots": they appear as bright coloured boxes (the brighter, the more discussed). Among the most discussed components of the software system are the graphical user interface, compiler, data binding and internal components such as the debugger.

Surprisingly, our visualization of "discussion hotspots" suggests that a substantial amount of discussion takes place on source code contained in the `examples` directory of the `SWT framework`. In order to empirically validate this "hotspot", we inspected a random sample of traceability links that point to files in this subsystem. Notably, we observed that developers in ECLIPSE appear to

extensively borrow from code fragments contained in issue reports for use in regression testing. These examples are saved in the form of code snippets, minimal stand-alone programs that demonstrate functionality such as API usage on the one hand, and act as a basis for different test cases on the other hand. Through the project's website[1], developers actively encourage users to contribute snippets through the `BugZilla` issue tracking system. Based on these observations, we conjecture that developers thus acknowledge the importance of code fragments contained in project discussions, and actively migrate code fragments into code snippets to be used for regression testing purposes.

5.6 *Conclusions*

Similar to previous approaches, fuzzy code search-based traceability linking has a specific intended use case, and limitations within other use cases. The main focus of our approach lies on linking issue reports to source code, with the aim of *locating code* that is talked about in project discussions within the body of source code of the software system. As the source code body of large software systems can easily contain thousands of files, establishing such traceability links is no trivial task.

Our proposed solution leverages an existing token-based clone detection tool, `CCFinder`, which was designed to efficiently locate all the occurrences of similar code fragments in a software system, as a readily available implementation of fuzzy code search.

Through a case study on the `ECLIPSE` software project, we discovered that fuzzy code search-based traceability linking shares only a small percentage of traceability links with the state-of-the-art approach to link issue reports to source code files: commit-log analysis. We find that a combination of the sets created by both approach results in a 20.01% increased in total

[1] http://www.eclipse.org/swt/snippets/, last accessed February 2014

traceability links between issue reports and source code. We thus see a potential application of fuzzy code search-based traceability linking for recovering missing links, in the same vein of work presented by Wu et al. [Wu2011].

We demonstrated an example application of our approach: identifying and visualizing the parts of the software system that are discussed the most in issue reports. During the analysis of this visualization, we discovered that developers extensively borrow code fragments from project discussions for use in regression testing. Additionally, we identified a variety of interesting side effects of our traceability linking approach. For instance, we observed many clone groups that contain code fragments from multiple different discussions, i.e., traceability links established by our approach do not only link projects discussions to the source code, but also discussions of different issue reports among each other. As we have seen in section IV-C, our approach discovers links from multiple issue reports to the same source code files. Within this context, we see a potential application of our traceability linking approach for the identification of related, or duplicate issue reports – a research avenue that we plan to explore in future work.

The objective of our proposed fuzzy code search-based approach is to find high quality traceability links between issue reports and source code. Based on the results of our analysis, we recommend the use of the most appropriate approach to linking communication data to source code for the respective use case, as each method generates traceability links that are in conceptually different classes. Both the quantitative and qualitative analysis of our approach suggest that there is still much room for improvement in this research area: about one third of issue reports needed to be discarded as a result of the limitations of the used clone detection tool. Hence, one major direction of our future work is to study the applicability of other fuzzy code search approach, for example approximate string matching techniques [Navarro1999], to find occurrences of extracted code fragments in the project's code base.

5.6.1 *Relevancy to this Thesis*

The ability to link developer communication to the parts of the software that is being discussed (i.e., the particular parts of the source code the discussion references) is integral for the study of our research hypothesis in later chapters of this thesis. In particular, we use the links between developer communication and source code in Chapter 6 to extract the risk of defects being present in the source code that is being discussed. Within that context, we use defect risk as a quality measure, similar to previous work (ref. Chapter 2).

Furthermore, we use links between developer communication and source code for the manual investigation of the management of code contributions in Chapter 7.

Part III

The Relationship between Developer Communication and the Quality and Evolution of the Software

Defective software has become increasingly expensive in industry. The United States National Institute of Standards and Technology estimated in 2002 that the annual cost of defective software amounts to up to 59.5 Billion US Dollars [Tassey2002]. As a result, software quality with respect to the number of software defects has become an increasingly relevant concern for practitioners and researchers alike. Past research has been very successful in modeling (i.e. explaining) software defects based on a technical view on the software.

Popular metrics such as the CK metrics suite [Chidamber1994] provide technical information about the source code of a software, such as size, complexity, or functional dependencies. In addition, research has identified changeability of the source code as one of the strongest explainers of software defects [Munson1998].

In this thesis, we hypothesize that the social component of software development might play a significant role with respect to software defects. Software development is a complex activity that requires the continued coordination between software developers to manage this complexity. We conjecture that social aspects of communication surrounding the coordination activities of developers may give valuable insight for understanding software quality.

Chapter 6: The Relationships between Communication and Software Quality. In this chapter, we develop a set of novel metrics along four dimensions of social aspects between developers and their communication surrounding discussions in issue repositories. We use these metrics in a case study on multiple versions of the IBM Eclipse and Mozilla Firefox software products. We find that statistical models based on our social metrics can explain software defects as well as traditional models based on technical information. Among the social metrics, we find that unexpected disruptions in the communication flows between developers is among the strongest explainers of software defects.

Apart from an increasing awareness of the importance of software quality as a factor of business success, industry has recently discovered open-source software as a business model. In

this business model may companies open up their internal, private software development activities, and invite source code contributions from volunteer developers who are part of the community surrounding the software product. This practice aims at increasing the product halo [Thorndike1920], and thus the worth of the business.

In this thesis, we hypothesize that communication plays a key role in the open-source business model. Based on Bass' diffusion model [Norton1987], which states that "participation begets more participation" we conjecture that building a healthy community is major factor for taking advantage of the halo effect. In particular, we argue that developer communication impacts the evolution of the software through code contributions submitted by volunteer developers who are part of the product halo.

Chapter 7: The Impact of Communication on the Evolution of a Software. In this chapter, we investigate the relationship between developer communication and the evolution of a software through code contributions from a community of volunteer developers from the product halo. We first derive a conceptual model of how code contributions are managed in practices through a systematic study of the available documentation of seven large open source software projects. Through a case study on the Linux kernel, and the Google Android software, we investigate the role of developer communication in the five phases of our conceptual model. We find that timely communication with volunteer developers plays a significant role as to whether contributions can be successfully integrated into the software. We observe that industry is actively changing their contribution management processes to provide faster feedback to volunteer developers, in order to prevent wasted developer resources due to the limited time that contributors are available to address potential issues with their contributions.

In both chapters of this part, we use the tools and techniques developed in Part II, for mining of communication data (Chapter 3), the extraction of technical information from that communication data (Chapter 4), and to link developer communication to those parts of the source code that is talked about (Chapter 5).

6

The Relationships between Communication and Software Quality

In the previous chapter we linked communication between developers to the parts of the software that they were discussing. We have demonstrated that discussions surrounding reports of software defects frequently mention particular parts of the software. Correcting software defects accounts for a significant amount of resources in a software project. To make best use of testing efforts, researchers have studied statistical models to understand in which parts of a software system future defects are likely to occur. By studying the mathematical relations between explanatory variables used in these models, researchers can form an increased understanding of the important connections between development activities and software quality. Explanatory variables used in past top-performing models are largely based on source code-oriented metrics, such as lines of code or number of changes. However, source code is the end product of numerous interlaced and collaborative activities carried out by developers. Traces of such activities can be found in the various repositories used to manage development efforts. In this chapter, we present statistical models to study the impact of social interactions in a software project on software quality. These models use explanatory variables based

on social information mined from the issue tracking and version control repositories of two large open-source software projects. The results of our case studies demonstrate the impact of metrics from four different dimensions of social interaction on post-release defects. Our findings show that statistical models based on social information have a similar degree of explanatory power as traditional models. Furthermore, our results demonstrate that social information does not substitute, but rather augments traditional source code-based metrics used in explaining software defects.

6.1 Introduction

In the foreword to "Why programs fail" [Zeller2009], James Larus, director of Microsoft's Customer Care Framework (CCF) project notes: *"If software developers were angels, debugging would be unnecessary[...]"* as an homage to the famous words by James Madison. With this line, Larus expresses a fundamental software engineering problem that sparks enormous research efforts: software contains defects, and fixing these defects is very costly – even more so, if they are discovered after the software has shipped.

To reduce maintenance costs, researchers have extensively studied two core areas in empirical software engineering: understanding and minimizing the cause of defects and building effective systems to predict where defects are likely to occur in the software system [Bird2009b]. Both research areas are intertwined: knowledge gained from understanding root causes can help in building better predictors [Schroter2006], and at the same time the study of prediction models provides cues for understanding the causes of defects, such as complex code change processes [Hassan2009]. Past work in defect prediction makes extensive use of product and process metrics [Purao2003], obtained from the source code of a software system, such as code complexity [McCabe1976], code change metrics [Munson1998], or inter-dependencies of elements in the code [Nagappan2007].

However, source code is the end product of a variety of collaborative activities carried out by the developers of a software. Lately, researchers started to realize that the intricacies of these activities such as social networks [Wolf2009], work dependencies [Cataldo2009b] and daily work routines [Sliwerski2005] impact the quality of a software product. Traces of these activities can be found in the repositories that developers use on a day to day basis, such as version archives, issue tracking systems, and email communication archives. In this study we investigate how we can use information about the social interactions in the community for defect modeling, and set out to study their relative impact on software quality. In particular, we focus

on social information extracted from discussions on issues reports, which are stored in issue tracking systems. We use statistical models to establish and inspect mathematical dependencies between defects and social interaction metrics – an approach that has been successfully used in previous research to study the relation between source code metrics and defects [Cataldo2009b; Hassan2009; Nagappan2005; Nagappan2007; Schroter2006; Zimmermann2007]. In particular, we set out to study the following relations:

(1) The relationship between the social structures, extracted from discussions in issue tracking systems and software quality, as expressed through post-release defects.

(2) The relationship between the contents and characteristics of communication, measured through metrics computed from issue tracking system discussions, and software quality, as expressed through post-release defects.

(3) The relationship between workflow in the community, expressed through activities in the issue tracking systems, and software quality, expressed through post-release defects.

Through case studies on the ECLIPSE and Mozilla FIREFOX software systems, we find that such relationships exist and that they can be used to create statistical models with explanatory power similar to traditional models based on product and process metrics. In addition, we find that a combination of our model based on social interaction metrics and a traditional model based on product and process metrics, yields higher explanatory power than each of the models taken separately.

6.1.1 *Organization of this Chapter*

The rest of this chapter is organized as follows. Section 6.2 describes the set of social information metrics we use throughout this study. Section 6.3 presents the general design of our case studies, together with a discussion of the statistical (regression) models and the methods used to describe their performance.

Section 6.4 presents our case study on the ECLIPSE software system, and section 6.5 presents our case study on the Mozilla FIREFOX software system. We discuss our findings of both case studies in more detail in Section 6.6, and perform a comparison of our observations across both projects. Section 6.7 investigates the possibility of combining models based on social information metrics with traditional models based on source code metrics. We close this chapter by discussing related research work (Section 6.8), possible threats to the validity of our study (Section 6.9), and our conclusions (Section 6.10).

6.1.2 *Contributions*

The work presented in this chapter makes the following contributions to the research field. (1) We distill those metrics of social interaction, that are connected to software quality, and describe their effect through odds ratios. (2) We demonstrate that social information metrics complement traditional, source code based metrics, and investigate which of the social information metrics could be valuable for defect modeling. (3) We show that developer communication is a strong explainer of software quality.

6.2 *Social Interaction Metrics*

In this section we describe the four dimensions of social interaction metrics that we use in our statistical models. To help our readers follow along in the text and increase the readability of this work, we present an overview of these metrics in Table 6.1. For each metric, we briefly motivate its inclusion and outline our approach to measure it. Our social interaction metrics are determined from traces of activity that developers and users leave behind in the issue tracking system. Hence we calculate each metric on a per-issue report level.

We study the relation between social interaction metrics and software quality on a per-file level. As a result, we need to aggregate values across all issue reports associated with a file. Our default method of aggregation is to take the *average*. However, for some metric we are interested in their variability and for these metrics we will use *entropy* for aggregation. Entropy is a concept we borrow from information theory [Shannon2001]. The normalized entropy is defined as:

$$H(P) = -\sum_{k=1}^{n}(p_k \cdot log_n(p_k))$$

where $p_k \geq 0, \forall k \in 1,\ldots,n$ and $\sum_{k=1}^{n} p_k = 1$. Normalized entropy is an extension to Shannon's classical measure of entropy [Shannon2001] and allows us to compare entropy metrics across different distributions. Measures of entropy have been used in previous research to study the evolution of code changes [Hassan2009], noting that when a project is not managed well, or the code change process is not under control, the system will be in a state of maximum entropy (chaos). Through measures of the entropy of the social processes surrounding code changes, we are studying whether this conjecture holds true within our context.

To illustrate normalized entropy, we consider the following example (presented in Figure 6.1). Let m_A be a set of measures of the time (in hours) between the submission of consecutive discussion messages on issue *A*, and let m_B be a set of measures of the time (in hours) between the submission of consecutive discussion messages on issue *B*. Suppose we find that $m_A = \{1.1, 1.2, 1.3, 1.4\}$ and $m_B = \{1.1, 1.2, 4.0, 1.3\}$. Both sets of measures are the same, except that for one value: in m_B, the third measure turns out to be 4.0 hours between two messages. After normalization of measures, we obtain the sets of normalized values $\bar{m}_A = \{0.22, 0.24, 0.26, 0.28\}$ and $\bar{m}_B = \{0.14, 0.16, 0.17, 0.53\}$. We can now calculate the normalized entropy for both sets of measures, as presented in Equation 1, and find that $H(\bar{m}_A) = 0.9971$ and $H(\bar{m}_B) = 0.8735$. In particular, we want to note the following: the set m_B containing a larger variability of measures, but exhibits a lower measure of normalized entropy.

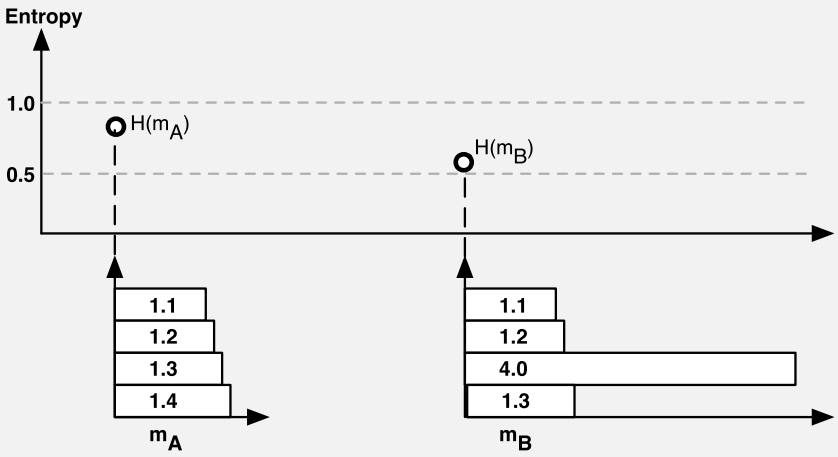


Figure 6.1: Example of entropy: a larger variability in the data leads to a lower measure of normalized entropy.

In general, we achieve the maximum value of entropy, if all elements in the set have equal values; and we achieve minimum entropy, if all except one measure have a value of zero.

6.2.1 *Dimension One: Discussion Content*

In this section, we describe the seven discussion content metrics that we use in our study. We choose to incorporate metrics on code examples, patches, stack traces and links found in discussions, to better understand the content of discussions and their impact. For example, Bird et al. note that technical talk (indicated by the presence of a larger amount of technical information items) can have a different impact than regular chitchat [Bird2009b]. We use the `infoZilla` tool [Bettenburg2008b] to extract technical information items from the textual contents of bug report discussions.

In a previous study [Bettenburg2008c], we asked developers of the ECLIPSE and MOZILLA projects, which of the information inside bug reports is most helpful for them when working on the reported issues. Among the top answers were information items like crash reports (in

the form of stack traces), source code examples and patches. As a precise understanding of the underlying defect is crucial for addressing a reported issue adequately, we conjecture that the presence or absence of information items in bug reports can possibly influence the quality of the source code changes carried out under the context of the reported issue. For example, discussions of test cases can help developers to recover the rationale and intended correct behavior of a software system.

Source Code Examples (Amount, Complexity)

Our first metric is the *amount of source code* (NSOURCE) present in a discussion. Source code can find its way into an issue report due to several reasons: reporters point out specific classes and functions they encountered a problem with, or provide smaller test-cases to exactly illustrate a misbehavior; developers point users at locations in the source code they require more information about, and discuss possible ways to address an issue with peers. In particular, we are measuring the number of complete code examples, rather than lines of code, since source code often loses its original formatting when present in discussions. Our `infoZilla` tool reports source code examples, as the largest blocks of source code surrounded by either natural language text, or other structural elements. As the complexity of the code discussed might be an indicator for the intricacy of the reported problem and an indicator of future risk, we also compute the *complexity of the discussed code* (NSCOM) as a qualitative measure for each of the source code examples., we use McCabe's cyclomatic complexity [McCabe1976], rather than lines of code as our complexity metric.

Our computation of McCabe complexity is analog to the implementation found in the open source static code checker PMD[1] and is defined as

$$McCabeComplexity = 1 + count(DecisionPoints, code)$$

with a decision point being either an `if`, `else`, `else if`, `for`, `while`, `do`, or `case` statement in the source code.

[1] http://pmd.sourceforge.net/

Patches (Amount, Filespread)

Our second metric is the *amount of patches* (NPATCH) provided in the discussions. Publicly discussed patches provide peer-reviewed solutions to the reported issues. Multiple patches can either present different solutions to the same problem, solutions a variety of less complex subproblems, or be different revisions of the same solution that has been refined through the discussion. In addition to the quantitative measure of patches we also compute a qualitative measure by recording the *number of files changed by a patch* (PATCHS). Through this metric we capture the spread of a patch. We motivate this choice with the idea that patches resulting in large or wide-spread changes to the source code might negatively impact dependent parts of the code (even though they might correctly fix the reported issues) [Hassan2009].

Stack Traces (Amount, Stacksize)

Our third metric records the *amount of stack traces* (NTRACE) provided in the contents. Information inside stack traces provides helpful information for developers to narrow down the source of a problem, and are hence valuable for finding and fixing the root causes of issues rather than addressing their symptoms [Zeller2009]. We use the number of methods reported in the stack traces as a qualitative measure for the *size of stack traces* (TRACES).

Links

Our fourth metric of discussion contents records the *amount of links* (NLINK) present. Developers and users use URLs to provide cross-references to related issues and to refer to external additional information that might be relevant to the original problem. We make no distinction between internal links (e.g., to other issue reports) and external links (e.g., to third-party websites).

6.2.2 *Dimension Two: Social Structures*

In addition to the information obtained from the textual contents of discussions, we compute a number of metrics to describe the social structures between developers and users, created through issue report discussions. In the following we describe the five metrics of social structures used in our study.

Discussion Participants

In order to contribute to the BUGZILLA system, users have to sign in with a username and password. The username acts as a unique handle for all activity in the issue tracking system. We conjecture that the total amount of unique participants in the discussion of an issue report is an indicator of the relative importance of the reported problem. Our first measure hence counts the *number of unique participants* (NPART) in the discussion.

Role

In this study we further categorize participants into two different roles: developers and users. We consider a participant to be a developer, if he was assigned to fixing at least one BUGZILLA issue in the past. By measuring the *number of unique users* (NUSERS) and the *number of unique developers* (NDEVS) participating in the discussions we can distinguish between internal discussions (more developers than users), external discussions (more users than developers) and balanced discussions (even amount of developers and users).

Experience

Another social property of participants, orthogonal to their role, is their degree of experience in the community. In our study, we determine the top three participants with the highest experience (expressed by the past amount of contributed messages) for each discussion attached to an issue report. These measures are captured in the three variables (CON1), (CON2), and (CON3). In

particular, each variable CON1, CON2, and CON3 contains the unique Bugzilla login names of the three developers, we determined to be the most experienced. The degree of experience of a participant can influence the development process connected to an issue; for example Guo et al. show that defects reported by more experienced users have a higher likelihood to get fixed [Guo2010].

Centrality

Our last metric of social structure (SNACENT) is taken from the area of social network analysis, called *closeness-centrality*. This metric is commonly used in social network analysis to describe the efficiency of spreading information among a group of people [Wasserman1994]. We conjecture, that inefficient relay of crucial information might have a negative impact on software quality. We measure closeness-centrality as follows. For each discussion attached to an issue report in the bug database we first construct a discussion flow graph [Mertsalov2009]. The discussion flow graph is an undirected graph that has participants as nodes and contains an edge for every pair of two consecutive messages in the discussion, connecting both message senders. We express the interconnectedness of nodes (participants) in the discussion flow graph as a measure of [Wasserman1994].

The closeness-centrality C_C of each participant in a discussion is the inverse of the average shortest-path distance d_S from the participant in the discussion flow graph to every other participant in the graph. Figure 6.2 illustrates an example discussion and corresponding discussion flow graph. In this example, participant A is connected to participants B and D directly. Participant C is connected only to participant B, and participant E is connected only to participant D. The closeness-centrality for participant E is

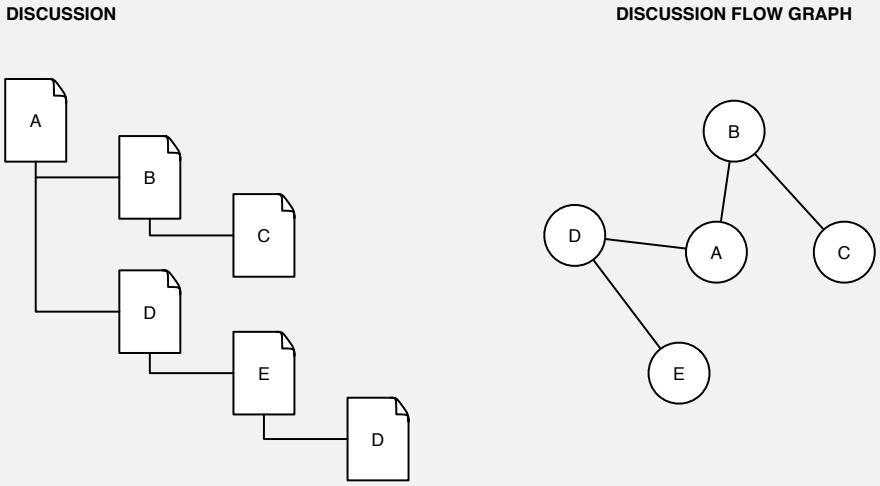


Figure 6.2: Example discussion and resulting discussion flow graph.

$$
\begin{aligned}
C_C(E) = \quad & (\tfrac{1}{4} \cdot (d_S(E,A) + d_S(E,B) + d_S(E,C) + d_S(E,D)))^{-1} \\
= \quad & (\tfrac{1}{4} \cdot (2+3+4+1))^{-1} \\
= \quad & \tfrac{2}{5}
\end{aligned}
$$

Similarly, the closeness-centrality value for participant A is $\frac{2}{3}$, and for both participants B and D it is $\frac{4}{7}$. Since closeness-centrality is a per-node measure (one value for each discussion participant, and one such set of values for each discussion associated with a particular file), we aggregate the closeness-centrality of all participants in the discussion into a single value, through normalized entropy. The more participants are equally able to spread information to everybody else, the higher the normalized entropy measure will be.

6.2.3 *Dimension Three: Communication Dynamics*

In addition to information about discussion content and involved participants, we attempt to measure discussion activity both quantitatively and qualitatively. In this context, we refer to

communication dynamics as the changing attributes of a discussion as it unfolds when new discussion activity is added. In the following we describe the six metrics of communication dynamics, used in our study.

Number of Messages

By their very nature, issues that are complex, not well understood, or controversial require a greater amount of information exchange relative to simple issues. We capture this intuition through a measure of the *amount of messages* (NMSG) exchanged in a discussion.

Length of Messages

Following the same intuition, we define two additional metrics: first, the *number of words in a discussion* (DLEN), and second *discussion length entropy* (DLENE), as a proxy to the variability in wordiness in discussions. We consider the "wordiness" of messages as an indicator for the cognitive complexity of the reported issue and greater fluctuations of wordiness (resulting in a higher measure of entropy) as an indicator for possible communication problems.

Reply Time

Cognitive science defines communication as "the sharing of meaning" [DEste2004; Alatis1993]. The absence of communication for an extended period of time, or distorted communication, is related to misinterpretations and misunderstandings. In the context of software development, such misinterpretations when carrying out changes to the source code can be the cause of errors. We capture this idea by measuring both, the *mean reply time between messages* (REPLY), and the *reply time entropy* as a proxy to the variability in reply times (REPLYE) in discussions. We conjecture that a higher variation in reply times (e.g., a long pause in an otherwise fast-paced discussion) captures temporal anomalies in the discussion flow that might indicate potential problems, and thus possibly post-release defects.

Interestingness

The BUGZILLA system allows users to get automatic notifications when an issue report is changed, via so-called "CC-lists". We use a measure of the number of people who signed up for such notifications as an indicator of the *interestingness* (INT) of an issue report. This measure is different from the *number of participants*, since users of the issue tracking system can be on the notification list, while not contributing to the discussion for an issue report. In addition we capture the variability of interestingness in a measure of *interestingness entropy* (INTE). We conjecture that a larger variability of interestingness (e.g., a rather uninteresting file suddenly becomes very interesting, versus a file that is always rather uninteresting), might thus be an potential indicator for post-release defects.

6.2.4 *Dimension Four: Workflow*

Issue reports represent work items for developers and follow a set of states from creation until closure [Anvik2006] and transitions between these states create a workflow. Any workflow activity associated with each report is recorded in the BUGZILLA system. We conjecture that high workflow activity indicates work anomalies [Shihab2010b; Jeong2009; Guo2011], such as re-assignment of the work item to another developer, or re-opening reports that were previously marked as completed. To capture workflow activities, we measure the *total amount of workflow activity* (WA) associated with each issue report, as well as the *entropy of workflow activity* (WAE) to capture variability in the process.

Measure	Description
	Baseline Model
CHURN	Code churn is defined as the number of lines added, modified, and deleted between two consecutive versions of a source code file.
	Discussion Contents
NSOURCE	Number of source code regions found in a discussion by the infoZilla tool.
NSCOM	Average cyclomatic complexity of the code found in a discussion.
NPATCH	Number of patches found in a discussion by the infoZilla tool.
PATCHS	Number of files modified by a patch.
NTRACE	Number of stack traces found in a discussion by the infoZilla tool.
TRACES	Size of a stack traces in number of stack frames.
NLINK	Number of URL links to resources outside the discussion.
	Social Structures
NPART	Number of unique participants in a discussion.
NUSERS	Number of unique participants in a discussion, who are users.
NDEVS	Number of unique participants in a discussion, who are developers.
CON1-3	Unique login names of the three most experienced developers participating in a discussion.
SNACENT	The degree to which each participant talks to other participants, captured through a measure of clonesness-centrality.
	Communication Dynamics
NMSG	Total number of messages exchanged in a discussion.
DLEN	Number of words in a discussion.
DLENE	Variability in the number of words across all discussions on issue reports associated with a source code file.
REPLY	Mean reply time between the messages of a discussion.
REPLYE	Entropy of the mean reply time between discussions on issue reports associated with a source code file.
INT	Interestingness of an issue report, captured through the size of notification list.
INTE	Variability in the interestingness across all issue reports associated with a source code file.
	Workflow
WA	Workflow activity recorded for an issue report.
WAE	Variability in workflow activity across all issue reports associated with a source code file.

Table 6.1: Reference of the measures of social interaction used in this study.

6.3 *Study Design*

Our analysis uses statistical models to investigate the relation between social information and software quality. We do so by exploring the statistical relations between the failure proneness of files and the social information metrics captured from issue reports associated with these files. In this section we describe the design of our study, our model-building process and the results of our comparative analysis between the model based on social information metrics and classical models using code-based product and process metrics. Following previous work in defect prediction [Nagappan2007], we divide the collection of measurements into two distinct phases. For a period of 6 months before a release of the software we capture the social interaction metrics described in Section 6.2 for each file that has at least one issue report associated with it. We then measure the amount of defects (POST) reported for each file for the next 6 months following the release. For both case studied, we choose to perform our measures for periods of 6 months surrounding the releases of `Eclipse 3.0`, `Eclipse 3.1`, and `Eclipse 3.2`, as well as Mozilla `FIREFOX 1.5`, Mozilla `FIREFOX 2.0`, and Mozilla `FIREFOX 3.0`. From the measurements obtained for the corresponding time periods, we create regression models that set the occurrence of *post-release defects* into relation of our *pre-release measures*. The complete regression model has the following form:

$$
\begin{aligned}
Prob(Postrelease \quad Defects) = \theta \cdot CodeChurn \\
&+ \sum_i \alpha_i \cdot DiscussionContentsMetric_i \\
&+ \sum_j \beta_j \cdot SocialStructuresMetric_j \\
&+ \sum_k \gamma_k \cdot CommunicationDynamicsMetric_k \\
&+ \sum_l \delta_l \cdot WorkflowMetric_l + \epsilon
\end{aligned}
$$

Here, ϵ is called the *intercept* of the model, and $\theta, \alpha_i, \beta_j, \gamma_k$, and δ_l are called the *regression coefficients*. Based on this model, we will investigate the statistical relationships between the social interaction metrics of each dimension, which are used as the *regression variables* in the model, and the probability of post release defects, which is used as the *dependent variable* in the model. We start with a preliminary analysis of the regression variables using descriptive statistics, to illustrate general properties of the collected metrics. Next, we perform a correlation analysis to consider possible inter-relations between measurements. We then construct several logistic regression models to investigate the relative impact of each of the four dimensions of social interactions metrics on the risk of post-release defects. Our approach is similar to the work by Cataldo and Mockus [Cataldo2009b; Mockus].

We follow a hierarchical modeling approach when creating all our models: we start out with a baseline model that uses code churn, a classical defect predictor, as the regression variable. We then build subsequent models to which we step-by-step add our content, structure, communication dynamics and workflow metrics, and report for each model the explanatory power, χ^2, of the model. The χ^2 statistic can be thought of as a measure of "goodness of contribution from the set of regression variables" [Cohen2003]. In addition, we report for each model M_i the percentage of deviance explained by the model, which is a quality of fit statistic and is defined as

$$D(M_i) = -2 \cdot LL(M_i)$$

with $LL(M_i)$ denoting the log-likelihood of the model, and the deviance explained as a ratio between $D(M_0) = D(Defects \sim Intercept)$ and $D(M_i)$. This statistic is similar to the coefficient of determination, R^2, and describes the variability in the data set the model accounts for [Steel1960], and thus describes how well the model, when used for prediction, describes future outcomes. Our choice of χ^2 over R^2 as a measure of goodness is rooted in two observations. First, in logistic regression models, R^2 needs to be approximated through a pseudo-R^2 measure, whereas χ^2 is directly accessible. Second, as we add more regression variables to our logistic regression models through the hierarchical approach taken, the pseudo-R^2 measure increases, even if the added variables add no value to the regression models.

Overall, a hierarchical modeling approach has the advantage over a step-wise modeling approach that it minimizes artificial inflation of errors, and thus over-fitting [Cataldo2009b]. To determine the contribution of each dimension of social metrics to our model, we test for each subsequent model whether the difference in explanatory power from the previous model is statistically significant, and present the corresponding p-level.

To ease readability and interpretability of our results, we present the odds ratios [Edwards1963] of each regression coefficient, rather than the raw β-coefficients. For a presentation of the β-coefficients of each model, we refer our reader to Appendix B. Since logistic regression models express regression coefficients as the log of odds, the relationship between odds ratios (OR) and regression coefficients (β) is expressed through:

$$OR_i = e^{\beta_i} \qquad and \qquad \beta_i = log(OR_i)$$

An odds ratio greater than one indicates a positive relation between the dependent variable (risk of post-release defects) and the independent variables (social interaction metrics), whereas an odds ratio between zero and one indicates a negative relation. As we are working in a log-transformed space, the odds-ratios have to be interpreted accordingly: a single unit change in the

log-transformed space corresponds to a change from 1 to 2.71 ($= e^1$) units in the untransformed space.

As a word of warning, we want to note that odds ratios should not be compared directly. Odds ratios describe the effect of a one-unit increase of the regression variable on post-release defects, while keeping every other regression variable constant. However, since regression variables attain different actual values in both projects their relative effects might be different across projects. As an example consider the following imaginary regression variable X. If we would find that X has an odds ratio of 1.5 in ECLIPSE, but an odds ratio of 2.0 in FIREFOX, one might be tempted to argue that X has a larger effect on defects in FIREFOX than in ECLIPSE. Yet, it is possible that X only attains actual values between 0.1 and 0.5 in FIREFOX, whereas it could attain actual values between 1.0 and 2.0 in ECLIPSE, thus the true effect of X on defects is considerably larger for ECLIPSE than FIREFOX – despite the smaller odds ratio!

6.4 Case Study One: Eclipse IDE

6.4.1 Data Collection

For our case study on the ECLIPSE project, we used two main sources of available data for ECLIPSE. First, we obtained a copy of the project's BUGZILLA database. This database collects modification requests that are submitted electronically by a reporter. These requests are commonly referred to as "bug reports". However, we find this term misleading as not all reported issues are defects [Antoniol2008] and for the remainder of this paper we will refer to bug reports as "issue reports". Every report contains a variety of supporting meta-information such as a unique identification number, the software version, operating system, or the reporter's perceived importance. In addition, entries contain a short one-line summary of the issue at hand, followed

by a more elaborate description. After submission, entries are discussed in more detail between developers and users, who provide further comments. In our data, we treat the initial description written by the reporter as the first message, starting a discussion. Overall, we collected a total of 300,000 issues submitted to the BUGZILLA system between October 2001 and January 2010.

The second data source we use is the source code archive of the ECLIPSE project. We obtained a snapshot of the CVS software repository, which contains the project's source code, as well as all the information about past changes that have been carried out by developers. To record which files were changed together in the form of a transaction, we perform a grouping of single change records using a sliding window approach [Sliwerski2005]. Overall, we collected 977,716 changes (accounting for 224,643 transactions) carried out between October 2001 and December 2009.

In order to link information from both repositories together, we automatically inspect the transaction messages to identify pointers to issue reports. Each number mentioned in a transaction message is treated as a potential link to an entry in the bug database. Our algorithm initially assigns low trust to each potential link, but this trust increases when we find additional clues of the link's validity, such as keywords like "bug" or "fix", or common patterns used to mark references like "#" followed by a number. This approach was used in previous research [Cubranic2003; Schroter2006; Sliwerski2005; Fischer2003a] with high success. To further increase the quality of links, we incorporated the improvements by Bird et al. [Bird2009b]. Through these links we can then associate issue reports with files. Overall, we were able to establish 67,705 such links.

We used these links to compute social interaction metrics for the 6 months periods before the three major releases of ECLIPSE 3.0 (released June 21st, 2004), ECLIPSE 3.1 (released June 28th, 2005), and ECLIPSE 3.2 (released June 30th, 2006), and to count the amount of post-release defects for the 6 month periods after each release. To help our reader following along with our case study, we first present a detailed analysis and interpretation of our statistical models for ECLIPSE 3.0, followed by a discussion and comparison of models for releases 3.1

and 3.2 at the end of the section.

6.4.2 *Preliminary Analysis of Social Interaction Measures for ECLIPSE 3.0*

	Mean	SD	Min	Max	Skew
POST	1.16	2.28	0.00	35.00	5.00
NSOURCE	0.86	2.48	0.00	48.00	7.14
NSCOM	0.27	0.49	0.00	5.00	2.77
NPATCH	0.02	0.24	0.00	5.00	17.17
PATCHS	0.01	0.11	0.00	3.00	13.26
NTRACE	0.14	0.44	0.00	9.00	7.82
TRACES	3.56	10.73	0.00	175.00	5.04
NLINK	0.20	0.91	0.00	8.00	7.02
NPART	3.61	3.89	1.00	40.00	7.48
NDEVS	2.94	1.46	1.00	12.00	2.78
NUSERS	0.67	2.81	0.00	28.00	8.44
SNACENT	0.19	0.07	0.00	0.51	0.43
NMSG	7.32	5.92	2.00	67.00	3.13
REPLY	122.32	206.99	0.00	3239.00	5.17
REPLYE	0.10	0.09	0.00	1.00	1.29
DLEN	337.00	441.75	2.00	6259.00	4.60
DLENE	0.23	0.10	0.00	1.00	0.08
INT	3.80	8.42	0.00	55.00	4.94
INTE	0.14	0.26	0.00	1.00	1.74
WA	9.33	6.36	0.00	49.00	1.68
WAE	0.17	0.19	0.00	1.00	0.65

Table 6.2: Descriptive Statistics of Social Interaction Measures for ECLIPSE 3.0.

Our four dimensions of social interaction metrics (content, structure, dynamics, and workflow) represent different characteristics of collaborative activity on issues. While content metrics are more explicit in capturing the information exchanged between developers and users, our metrics of social structures are more implicit and capture the latent relationships and roles of stakeholders. Table 6.2 presents a summary of our metrics in the form of descriptive statistics.

Due to a relatively high amount of skew, we apply a standard log transformation to each social interaction measurement to even out the skewing effects during modeling [Bland1996].

Figure 6.3 summarizes the pairwise correlations between our 20 regression variables and our dependent variable in a correlogram visualization [Friendly2002]. A correlogram reports for each unique pair of variables the strength of the correlation as a color-coded field (red for positive correlation, blue for negative correlation) and the *p*-level at which the correlation is significant. This visualization technique allows us to identify "hotspots" that need our attention.

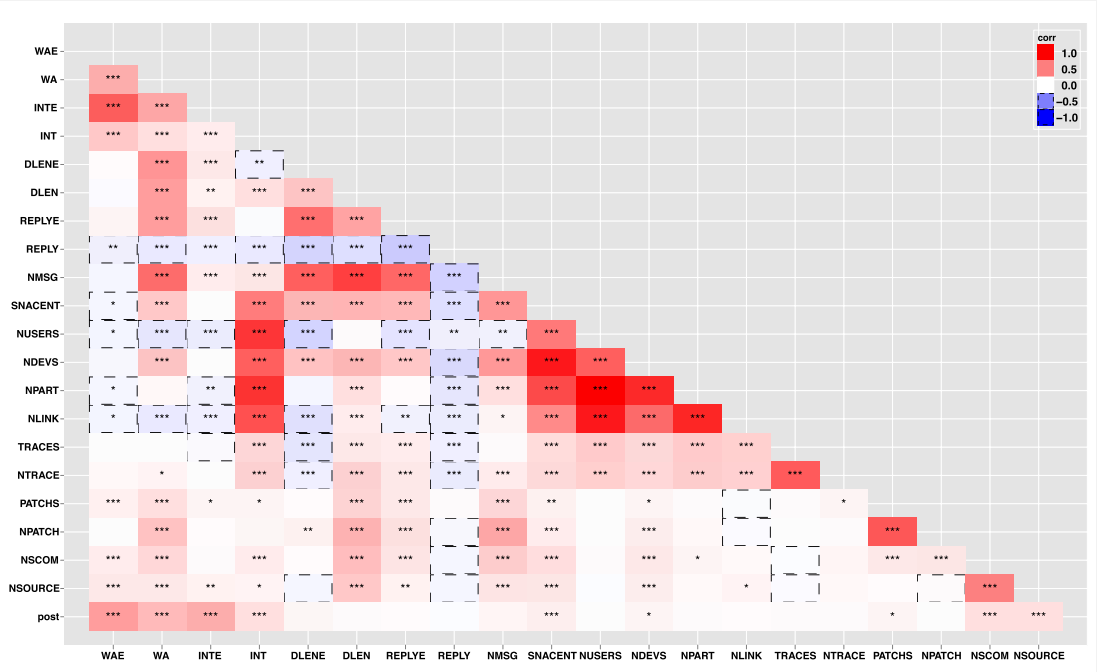


Figure 6.3: Pairwise correlations of social interaction metrics in ECLIPSE 3.0. The strength of correlations is indicated by fill intensities; negative correlations are marked with a dashed outline. The row labeled "post" refers to post-release defects. Stars denote the p-level as follows: * p<0.05, ** p<0.01, *** p<0.001.

We identify a number of intercorrelations between metrics from different dimensions in our dataset that could pose problems in our statistical modeling. For example, the measure of interestingness (INT) has a moderate to high correlation with our measures for number of users (NUSERS), number of participants (NPART), number of developers (NDEV), and number of links (NLINK). These correlations are statistically significant at p< 0.001. The first correlation between

interestingness (INT) and the number of participants (NPART) stems from a default setting in the ECLIPSE issue tracking system, which puts contributors automatically on a notification list for any updates to the issue. Some of the observed correlations can be explained by a certain inherent amount of redundancy in the collected data. For example, the number of participants (NPART) is highly correlated with the number of users (NUSERS) and number of developers (NDEVS). However, our motivation for incorporating such redundancy is to investigate whether splitting up the information into more specialized representations helps to improve our model. The same intuition holds for the measure of centrality (SNACENT).

	Variance Inflation Factor		
log(Y_i)	**Model 1**	**Model 2**	**Model 3**
NSOURCE	3.38	3.38	3.40
NSCOM	3.34	3.34	3.36
NPATCH	3.94	3.88	3.90
PATCHS	3.84	3.82	3.84
NTRACE	4.62	4.60	4.57
TRACES	4.78	4.75	4.70
NLINK	2.24	2.22	1.90
NDEVS	9.32	9.27	1.91
NUSERS	4.55	4.54	2.30
SNACENT	10.66	**10.65**	—
NMSG	**11.63**	—	—
REPLY	1.17	1.17	1.17
REPLYE	2.04	1.91	1.90
DLEN	4.21	1.91	1.87
DLENE	4.65	1.98	1.96
INT	2.82	2.82	2.60
INTE	1.71	1.71	1.71
WA	2.26	1.99	1.96
WAE	2.08	2.06	2.02

Table 6.3: Step-wise analysis of multicollinearity in the ECLIPSE 3.0 dataset. Numbers marked in bold font describe the highest variance inflation factors (VIF) at every step of our multicollinearity reduction process. These are the variables that are removed from the model in each step.

Since we observe a substantial number of high correlations among regression variables, we have to examine potential issues due to multi-collinearity among the variables. Even though the reliability of the statistical model as a whole is not affected by multi-collinearity issues, strong

correlations among independent variables often leads to an error in estimates of the regression coefficients that we later use to investigate the relative impact of each variable on post-release defects. One widely adopted approach to reducing multi-collinearity is Principal Component Analysis (PCA). However, in the context of our research goal, PCA is a poor choice due to a disadvantageous side-effect of the approach [Shihab2010a]. The result of PCA is a new set of regression variables, or principal components, which are linear combinations of all the input variables. As a result, we can not analyze the effects of the original regression variables anymore, which is the very purpose of this study. As an alternative way to address the problem of multi-collinearity, we perform a stepwise refinement of the set of variables we use through measuring the variance inflation factors for each variable. Variance Inflation Factors (VIF) are widely used to measure the degree of multi-collinearity between variables in regression models [Kutner2004].

Since the cut-off value for variance inflation factor analysis is a heavily disputed topic throughout statistical literature, we decided to follow the example by Kutner et al. [Kutner2004], and remove those variables from the model that have a variance inflation factor greater than 10. We start our analysis with a regression model that contains all our variables. The VIFs for this model are presented in Table 6.3, Model 1. We observe two variables that have a VIF greater than 10. We remove the highest one (NMSG) from the regression model and recompute the VIFs with the reduced set of variables. The resulting model, (Model 2 in Table 6.3) contains only one more variable with a VIF larger than 10. We remove the regression variable (SNACENT) from the model and recompute the inflation factors. In the resulting model (Model 3 in Table 6.3), no variables have a VIF larger than 5 and we finish our analysis of multicollinearity.

6.4.3 *Hierarchical Analysis*

After having determined the reduced set of regression variables with low multicollinearity, we proceed by investigating the relative impact of each of the four dimensions of social interaction metrics on the post-release defects.

The results of our hierarchical analysis are presented in Table 6.4. We start our hierarchical analysis with a *baseline model* which relates code churn (number of lines added, deleted, or modified in a file from one version to another) [Munson1998] to post-release defects. Code churn has been shown in the past to be one of the best code-based predictors of defects [Nagappan2005; Nagappan2007], even when used across projects [Zimmermann2009]. We obtained a measure of churn by mining the change histories of each file in the project's version control system. The results for the baseline model are presented in column MB of Table 6.4 and show that *CHURN* is positively associated to the failure proneness of a file during the post-release period. As expected, these results are in line with earlier findings [Nagappan2005; Nagappan2007].

Model M1 introduces the first dimension of social interaction: metrics concerned with the contents of issue report discussions. The results of the logistic regression model show that only specific structural information items are statistically significant. When looking at the odds ratios, we observe is a significant relationship between the number of files modified by patches (PATCHS) and future failure proneness of files. This result confirms earlier findings on the risk of scattered changes [Hassan2009].

The second observation we make is a positive link between the number of source code samples (NSOURCE) and future defects. This is surprising, as we initially expected code samples to have a beneficial effect (as explained earlier in the motivation of this metric). One possible explanation might be that developers trust user provided sample solutions and incorporate their proposed (yet possibly flawed) modifications without further verification; another explanation might be that code, which warrants an extended discussion, is more complex and thus more error-prone.

Furthermore, we observe a strong relationship between the number of links (NLINK) provided by users and failure proneness. One possible explanation for this relationship could be, that links to additional information (such as related issues) act as an indicator for hard-to-fix, or complex problems.

$\log(Y_i)$	MB	M1	M2	M3	M4	M5
CHURN	**4.996** ⋆	**4.631** ⋆	**4.658** ⋆	**5.303** ⋆	**3.688** ⋆	**4.470** ⋆
NSOURCE		**1.694** ⋆	**1.698** ⋆	**1.772** ⋆	**1.769** ⋆	**1.667** ⋆
NTRACE		0.79	0.768	0.864	0.881	1.115
NPATCH		**0.209** ∘	**0.210** ∘	**0.284** +	**0.231** ∘	0.291
NSCOM		1.218	1.194	1.246	1.208	1.244
PATCHS		**12.607** ∘	**12.626** ∘	**11.200** ∘	**12.736** ∘	**18.207** •
TRACES		1.016	1.012	1.004	0.989	0.975
NLINK		**1.764** ⋆	**1.613** •	**1.600** •	**1.666** •	**1.596** +
NPART			2.481	2.888	4.480	4.542
NDEVS			0.475	0.582	0.385	0.274
NUSERS			0.749	0.803	0.692	0.792
REPLY				1.019	0.986	0.982
REPLYE				**0.117** ⋆	**0.082** ⋆	**0.044** ⋆
DLEN				0.936	**0.898** ∘	**0.876** +
DLENE				2.499	1.251	2.044
INT				**0.829** •	**0.821** •	0.963
INTE				1.109	1.013	1.306
WA					**1.432** ⋆	**1.224** +
WAE					**2.718** ∘	2.169
CON1-3						**Fig. 4** ⋆
χ^2	**559.01** ⋆	**698.5** ⋆	700.15	**731.5** ⋆	**752.3** ⋆	**1055.19** ⋆
Dev. Expl.	10.71%	13.38%	13.41%	14.02%	14.41%	26.07%
$\Delta\chi^2$		139.48	1.652	31.357	20.28	302.87

⋆ p<0.001, • p<0.01, ∘ p<0.05, + p <0.1

Table 6.4: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics in ECLIPSE 3.0. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

Overall the results show that discussion content metrics are indicators of increased future failure proneness of a file. The explanatory power of the model increases by 2.67% over the baseline model and this increase is statistically significant.

Model M2 introduces the second dimension of social interaction metrics: information on social structures. The results show that the role of participants and the overall amount of participants in a discussion have no statistically significant impact on the future failure proneness of files. As a result we see no increase in the explanatory power of the model by introducing the role

of participants. We left out the measures of experience from this model, as we record them as factors with many levels that may disrupt our hierarchical modeling approach. We will revisit these measures later in model M5.

> *Overall, we cannot find a significant relation between the role of participants and post-release defects. The explanatory power of the extended model increases only marginally, however this increase is not statistically significant.*

Model M3 introduces metrics from the category of communication dynamics. The results show a statistically significant and strongly negative relation between the measure of reply time entropy and future failure likelihood, yet there exists no statistically significant relation between the related quantitative measure of average reply time length. The link between reply time entropy and failure proneness stays strong throughout the hierarchical process and indicates a relevant relationship. The second relation that we find is a moderately negative relation between the interestingness of an issue report and post-release defects. This relation however becomes irrelevant at a later point, when we introduce experience in model M5.

> *Overall, we observe a strong effect of discussion flow inconsistencies on the failure proneness of files associated with the discussion. Even though the explanatory power of the extended model increases by only 0.61%, this increase is statistically significant.*

Model M4 introduces the last category of social interaction metrics used in our study: workflow activity. Our results show a positive relationship (with respect to the odds ratios of the corresponding regression coefficients) between the total amount of workflow activities and post-release defects. In particular, increased workflow activity and workflow activity entropy are both connected to a substantial increase in post-release defects.

Our findings suggest that workflow activities play a marginal, complementary role in the relation between social interaction metrics and post-release defects. This is also indicated by the minor, yet statistically significant increase of explanatory power of the extended model (0.39%) when adding workflow activity metrics.

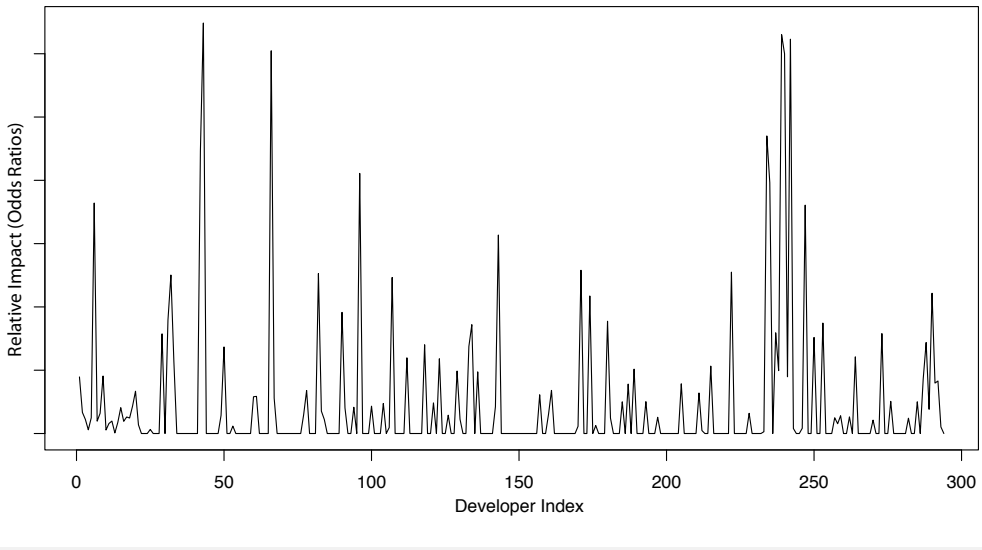


Figure 6.4: Odds ratios for experience metrics in model M5, each index represents one distinct level (developer) of the factor variables.

We revisit the dimension of social structures in Model M5, by adding our measures of experience. These measures are expressed as three factors (with the unique login handles of discussion participants as levels) and record the three most experienced contributors for each discussion. When used in a regression model, each factor generates a large quantity of binary variables (as many as it has different levels). As a consequence, we do not show the complete model containing all these binary variables. However, we measure the inherent effect of the factors on the statistical model, using a random-effect analysis of variance (often referred to as a type II ANOVA test), and present a plot of the odds ratios of each factor level in Figure 6.4. As contributors are uniquely identified in the ECLIPSE issue tracking system by their email addresses,

we do not include their names in this paper for privacy reasons; instead we anonymized each name by assigning a unique number. Our analysis of variance tests for the experience measures shows that they are statistically significant at $p < 0.001$. From the plot of odds ratios for each developer in Figure 6.4, we observe that certain experienced participants in a discussion are strongly related to the presence of post-release defects (indicated by the spikes of the relative odds ratios in the plot).

The increase in explanatory power of model M5 is over-proportionally large (compared to the effect of the previous four dimensions). As a result, we performed further analysis to determine, whether the inclusion of the experience metric leads to over-fitting in the statistical model (i.e., the effect captured in this metric describes random observations, rather than a real underlying relationship).

To judge possible over-fitting, we divide our complete set of data into a training set (90% of the data) and a testing set (10%) of the data. On both sets we train a Model M5 and compare the χ^2 values. We repeat this process ten times, with random 90/10 splits (often referred to as "10-fold cross validation"). In each of these 10 runs, we observe a large difference (corresponding to differences between 10.12% and 16.31% of deviance explained) in χ^2 values between the model obtained from the training set and the model obtained from the testing step. These large differences confirm that CON1-CON3 indeed lead to an over-fitting of the regression model and should thus not be included in the statistical model(s).

Overall, the experience metric increases the explanatory power of the extended model significantly. The increase of 11.66% is statistically significant. The plot of odds ratios for each developer determined as experienced suggests that particular contributors in a discussion act as an indicator for future failure proneness. However, we found that the inclusion of this metric leads to a severe over-fitting in the model. As a result, we need to exclude CON1-CON3 for the rest of this paper, and perform any comparisons between projects and releases based on model M4.

6.4.4 *Additional Versions of ECLIPSE*

$\log(Y_i)$	MB		M1		M2		M3		M4		M5	
CHURN	6.573	*	5.723	*	5.668	*	5.253	*	3.295	*	3.873	*
NSOURCE			1.340	*	1.239	•	1.338	*	1.314	*	1.378	*
NSCOM			0.613	*	0.597	*	0.612	*	0.586	*	0.651	•
PATCHS			1.552		1.709		1.678		1.799		2.612	
NPATCH			1.259		1.050		1.601		1.450		1.817	
NTRACE			0.701	+	0.643	∘	0.668	∘	0.730		0.778	
TRACES			1.094	+	1.123	∘	1.157	•	1.090	+	1.100	+
NLINK			0.274	*	0.270	*	0.252	*	0.240	*	0.352	*
NUSERS					3.480	*	4.240	*	5.040	*	3.887	*
SNACENT					0.254	*	0.143	*	0.212	•	0.246	∘
REPLY							0.972	+	0.917	*	0.958	∘
REPLYE							3.610	*	2.565	∘	2.514	∘
DLEN							0.761	*	0.717	*	0.753	*
DLENE							27.210	*	6.561	*	3.018	∘
INT							1.081	+	1.071		1.149	∘
INTE							1.545	•	1.144		1.376	+
WA									1.710	*	1.410	*
WAE									3.924	*	3.618	*
Chi Sq.	1643.9		2134.4	*	2434.5	*	2644.3	*	2744.0	*	4288.3	*
Dev. Expl.	11.66%		15.15%		17.27%		18.75%		19.46%		30.41%	
Delta Chisq			490.48		300.1		209.74		99.72		1544.2	

⋆ p<0.001, • p<0.01, ∘ p<0.05, + p <0.1

Table 6.5: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics for ECLIPSE 3.1. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

We repeated the same hierarchical modelling approach for two additional releases of ECLIPSE. The final models are presented in Table 6.5 and Table 6.6. During multi-collinearity analysis, we removed NDEV and NMSG in the case of ECLIPSE 3.1, and NPATCH, SNACENT, as well as NMSG in the case of ECLIPSE 3.2. As a result, the SNACENT regression variable was only available for building models of ECLIPSE 3.1, in which it was not deemed statistically significant. Overall, we observe a similar increase of explanatory power as in our detailed case study of ECLIPSE 3.0. Each introduction of the four dimensions of metrics adds a statistical significant (yet relatively small) amount of information to the models. Notably, in ECLIPSE 3.1 and ECLIPSE 3.2, the information added by the second dimension (social structures), is

log(Y_i)	MB		M1		M2		M3		M4		M5	
CHURN	4.614	*	4.277	*	4.312	*	3.332	*	4.867	*	5.809	*
NSOURCE			1.991	*	1.933	*	2.042	*	2.056	*	1.894	*
NSCOM			0.572	*	0.604	*	0.564	*	0.552	*	0.487	*
PATCHS			1.592		1.525		1.648		1.791		2.083	
NTRACE			0.581	∘	0.566	∘	0.690		0.678		0.604	
TRACES			1.223	*	1.225	*	1.167	∘	1.182	•	1.133	+
NLINK			1.254	*	1.223	•	1.136		1.056		1.543	*
NDEVS					0.882	∘	0.814	•	0.785	*	0.869	
NUSERS					1.186	•	1.126	+	1.134	+	0.866	
REPLY							1.002		1.029	∘	1.059	•
REPLYE							4.808	*	5.073	*	7.708	*
DLEN							0.858	*	0.904	*	0.948	
DLENE							0.305	*	0.541	+	0.150	*
INT							1.210	*	1.186	*	1.081	
INTE							2.580	*	3.415	*	3.148	*
WA									0.728	*	0.744	*
WAE									0.342	*	0.257	*
Chi Sq.	1198.83	*	1355.13	*	1364.31	∘	1470.4	*	1521.04	*	3106.11	*
Dev. Expl.	7.84%		8.86%		8.92%		9.62%		9.95%		20.31%	
Delta Chisq			156.3		9.18		106.09		50.64		1585.07	

⋆ p<0.001, • p<0.01, ∘ p<0.05, + p <0.1

Table 6.6: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics for ECLIPSE 3.2. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

deemed statistically significant, as opposed to ECLIPSE 3.0.

Overall, we find the following regression variables consistent across all studied releases: code churn (CHURN), number of source code examples (NSOURCE), number of files modified by patches (PATCHS), number of stack traces (NTRACE), discussion length (DLEN), and variability of interestingness (INTE). Among these metrics, CHURN, NSOURCE, PATCHS, and INTE are associated with an increased risk of post-release defects, whereas NTRACE and DLEN are associated with a reduced risk of post-release defects. Our findings confirm previous work on the relationship between code churn and defects [Nagappan2005], modification spread and defects [Hassan2009], as well as the helpfulness of stack traces when correcting defects [Schroter2010]. The observed relationship between the number of source code examples (NSOURCE) and post-release defects might be explained through the need for concrete examples when issues are more

complex, or hard to locate and fix. Further investigation of the relationship between variability in interestingness (INTE) and risk of post-release defects is needed before we can attempt an explanation, and is part of our future work.

6.5 Case Study Two: Mozilla Firefox

6.5.1 Data Collection

For our case study on the Mozilla FIREFOX project, we obtained a snapshot of the concurrent version control (CVS) system of the Mozilla platform, and a snapshot of the BUGZILLA issue tracking system. The CVS system contains the development history of the Mozilla platform up to (but not including) version 3.5 of the FIREFOX browser. For the development of FIREFOX version 3.5 and higher, the Mozilla team switched to the Mercurial distributed version control system.

Overall, we collected a total of 567,595 issues that have been submitted to the BUGZILLA system between April 1997 and August 2010. The collected version control history contains a total of 664,626 changes (accounting for 217,919 transactions) that have been carried out between April 1998 and August 2010. For the Mozilla FIREFOX project, crash logs were recorded in the form of *Talkback* traces between release 2.0 and 3.0. This proprietary crash reporting system was replaced in release 3.0 with an open-source version. These new crash logs are no longer collected in the issue tracking system. As a result, we have no stack trace measure available for our case study on the Mozilla FIREFOX project.

In order to link issue reports to transactions in the version control system, we use the same approach described in our case study on the ECLIPSE project (Section 6.4). However, we modified the set of keywords that point at bug identifiers to include those patterns that

	Mean	SD	Min	Max	Skew
POST	0.21	0.79	0.00	22.00	7.89
CHURN	2.74	4.58	1.00	83.00	6.67
NSOURCE	6.13	14.86	0.00	130.00	5.08
NSCOM	0.65	0.83	0.00	13.00	2.18
NPATCH	0.00	0.03	0.00	1.00	25.88
PATCHS	0.00	0.04	0.00	1.00	23.96
NLINK	2.04	3.38	0.00	31.00	5.11
NPART	5.90	3.71	1.00	41.00	2.32
NDEVS	5.39	3.09	1.00	33.00	1.91
NUSERS	0.51	1.09	0.00	19.00	4.42
SNACENT	0.26	0.08	0.00	0.50	-0.39
NMSG	19.21	15.74	1.00	137.00	1.91
REPLY	53.82	92.05	0.00	2246.00	9.17
REPLYE	0.21	0.11	0.00	0.64	0.11
DLEN	1212.47	1330.70	2.00	13325.00	2.45
DLENE	0.31	0.09	0.00	0.52	-0.85
INT	9.70	8.61	0.00	106.00	2.30
INTE	0.12	0.18	0.00	1.00	1.45
WA	24.31	16.90	0.00	86.00	1.11
WAE	0.09	0.14	0.00	1.00	1.32

Table 6.7: Descriptive statistics of Social Interaction Measures in the Mozilla FIREFOX project.

are specific to the Mozilla project. Following the methods described in Chapter 3, Chapter 4 and Chapter 5 of this thesis, we were able to recover 165,342 links between source code files and issue reports (and the discussions attached to these issue reports).

Similar to our case study on the ECLIPSE project (Section 6.4), we collected measurements for a period of six months before the releases of FIREFOX 1.5 (released November 29th, 2005), FIREFOX 2.0 (released October 24th, 2006), and FIREFOX 3.0 (released June 17th, 2008). For each release, we also collected the amount of post-release defects for a period of six months after release. In the following, we present a detailed case study on FIREFOX 3.0, which is followed by a discussion and comparison of earlier releases.

6.5.2 *Preliminary Analysis of Social Interaction Measures*

For our case study on FIREFOX 3.0, we follow the same statistical approach described earlier in our case study on the ECLIPSE project (Section 4.2) and begin with a general analysis of the data in the form of descriptive statistics, followed by an analysis of multi-collinearity. Table 6.7 presents a summary of our metrics in the form of descriptive statistics. We again observe a relatively high amount of skew in the data, such that we will use standard log transformations of all metrics in the remainder of our analysis.

The results of our analysis of pair-wise correlations are presented in Figure 6.5. We see that our data exhibits a high amount of multi-collinearity that we have to deal with before creating our statistical models. Especially notable are the observed correlations between the measures of the number of participants in a discussion (NPART, NDEVS, NUSERS) and most other metrics, as well as correlations between interestingness (INTE) and workflow (WAE), and the size of patches (PATCHS) and the number of patches submitted (NPATCH).

In order to resolve these multi-collinearity issues, we again perform a step-wise VIF analysis, starting with a regression model that contains all our measurements as independent variables and removing the variable with the highest VIF at each step, before re-evaluating our model.

The results of this analysis are presented in Table 6.8. Model 1 denotes our starting model, containing all variables. We observe that the number of messages in the discussion (NMSG) has the highest VIF value of 20.44. We remove NMSG from our model and re-evaluate the variance inflation factors (Model 2), at which point we observe the highest VIF measure of 19.07 for the number of patches submitted (NPATCH). We remove NPATCH from our model and re-compute all variance inflation factors. In Model 3, we observe two variables with a VIF measure above 10: number of developers (NDEV) and the measure of centrality in the social network (SNACENT), which both showed a high inter-correlation in the correlogram (Figure 6.5). We remove the variable with the higher VIF measure, NDEV at a value of 14.03, and re-compute variance inflation factors of the remaining variables (Model 4). As all remaining variables have a

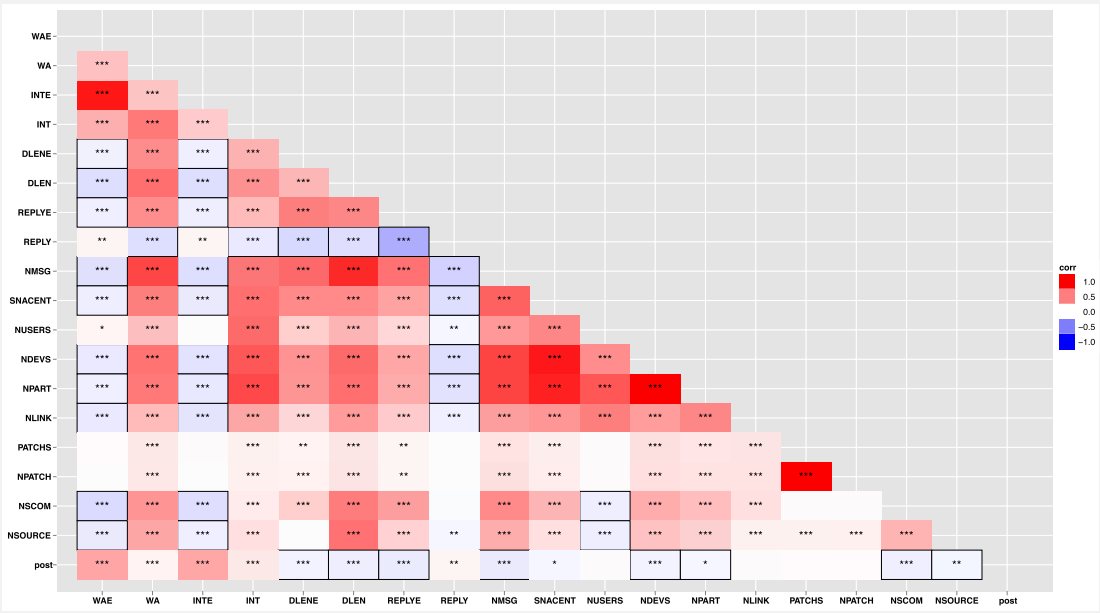


Figure 6.5: Pairwise correlations of social interaction metrics in Mozilla FIREFOX 3.0 with levels * p<0.05, ** p<0.01, *** p<0.001. Strength of correlations is indicated by colour intensities; negative correlations are marked with an outline. The row labeled "post" refers to post-release defects.

VIF measure below 10, we stop at this point and obtain our final set of variables with minimized multicollinearity.

6.5.3 *Hierarchical Analysis*

Using the reduced set of regression variables that we obtained by resolving multi-collinearity issues through step-wise VIF analysis in Section 6.5.2, we continue our hierarchical analysis to determine the relative impact of each of the four dimensions of social interaction metrics on post-release defects.

The results of our step-wise hierarchical analysis are presented in Table 6.9. Similar to our case study on ECLIPSE, we report all coefficients in the form of odds ratios, rather than the actual regression coefficients themselves, to ease interpretation.

	Variance Inflation Factor			
log(Y_i)	**Model 1**	**Model 2**	**Model 3**	**Model 4**
NSOURCE	2.90	2.87	2.87	2.86
NSCOM	2.60	2.67	2.67	2.65
NPATCH	18.81	**19.07**	—	—
PATCHS	18.75	19.00	1.05	1.03
NLINK	1.88	1.89	1.89	1.86
NDEVS	15.45	14.05	**14.03**	—
NUSERS	2.29	2.22	2.22	1.99
SNACENT	14.01	13.45	13.45	2.92
NMSG	**20.44**	—	—	—
REPLY	1.43	1.36	1.36	1.33
REPLYE	2.20	2.33	2.33	2.30
DLEN	8.91	3.01	3.01	2.99
DLENE	4.14	1.84	1.84	1.83
INT	3.73	3.71	3.71	3.40
INTE	5.74	5.92	5.92	5.88
WA	3.85	2.95	2.95	2.92
WAE	6.88	7.12	7.12	7.12

Table 6.8: Step-wise analysis of multicollinearity in the MOZILLA 3.0 dataset. Numbers marked in bold font describe the highest variance inflation factors (VIF) at every step of our multicollinearity reduction process. These are the variables that are removed from the model in each step.

Our hierarchical analysis starts with a *baseline model MB*, which relates only code churn to post-release defects. The results show, that churn is positively associated to post-release defects, same as it was in the ECLIPSE project.

Model M1 introduces the first dimension, structural information present in the discussions. Two information items turn out as statistically significant in the model: the amount of source code present in the discussion (NSOURCE), with an odds ratio of 0.891, and the amount of links (NLINK) with an odds ratio of 1.196. In contrast to our case study on ECLIPSE 3.0, NSOURCE is connected with an odds ratio smaller than 1.0, indicating that a larger amount of source code in the discussion is connected with a lower chance of post-release defects. A manual inspection of one hundred issue reports containing source code yielded no clear evidence on why this connection is opposite to our findings for ECLIPSE 3.0. On the other hand, the number

$\log(Y_i)$	MB		M1		M2		M3		M4		M5	
CHURN	**4.452**	⋆	**4.605**	⋆	**4.737**	⋆	**5.097**	⋆	**5.316**	⋆	**4.009**	⋆
NSOURCE			**0.891**	∘	0.904		0.943		1.018		0.915	
NSCOM			0.794		0.957		1.202		1.295		**1.599**	∘
PATCHS			4.974		8.498		6.168		**6.884**	∘	42.806	
NLINK			**1.196**	•	**1.378**	⋆	**1.508**	⋆	**1.475**	⋆	**1.253**	•
NPART					**0.177**	⋆	0.677		1.075		0.686	
NUSERS					**1.637**	⋆	**1.931**	⋆	**1.996**	⋆	**1.576**	•
SNACENT					**1208.985**	•	44.281		2.050		67.204	
REPLY							**0.928**	∘	0.987		1.000	
REPLYE							**0.016**	⋆	**0.067**	⋆	0.247	
DLEN							1.026		**1.117**	∘	1.043	
DLENE							0.814		**5.571**	•	1.150	
INT							**0.507**	⋆	**0.557**	⋆	0.846	
INTE							**3.061**	•	**3.969**	•	2.836	
WA									**0.492**	⋆	**0.715**	⋆
WAE									1.953		4.246	
χ^2	**744.14**	⋆	**773.88**	⋆	**807.54**	⋆	**933.29**	⋆	**1007.11**	⋆	**1836.43**	⋆
Dev. Expl.	14.90%		15.49%		16.17%		18.68%		20.16%		36.76%	
$\Delta\chi^2$			29.74		33.66		125.75		73.82		829.32	

⋆ p<0.001, • p<0.01, ∘ p<0.05, + p <0.1

Table 6.9: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics for Mozilla `FIREFOX 3.0`. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

of links in the discussion (NLINK), is connected with a larger amount of post-release defects, as it was in our case study on `ECLIPSE 3.0`. One possible explanation for this finding could be the presence of links to the external Talkback crash-report tracking systems present in the discussions of issue reports.

Overall, our results indicate that for `FIREFOX 3.0`*, source code in a discussion is connected with a decreased failure proneness, whereas the number of links is connected with an increase failure proneness. The explanatory power of the model increases by only 0.59% over the baseline model, however this increase is deemed statistically significant through analysis of variance (ANOVA).*

Model M2 introduces the second dimension of social interaction metrics: social structures. Similar to our findings in our previous case study on ECLIPSE 3.1 and ECLIPSE 3.2, we find a statistically significant connection of this dimension with post-release defects. All three variables, the number of participants in a discussion (NPART), the amount of participants that are considered users (NUSERS) and the centrality measure, which describes the degree to which participants communicate with everyone else in the discussion (SNACENT), are considered statistically significant for this model. Considering the odds ratios of each variable, we find that the overall number of participants in the discussion is connected with a decrease of post-release defects, whereas both the amount of users in the discussion as well as the degree to which each participant talks to everyone else, connected with an increase of post-release defects.

> *Overall, we find the second dimension, social structures to have a statistically significant effect on the explanatory power of our model. Even though the relative increase in explanatory power over model M1 is only 0.68%, this minor increase is statistically significant.*

Model M3 introduces the third dimension of metrics: communication dynamics. The results show a statistically significant and negative relation between the measures of reply time (REPLY) and the corresponding entropy measure (REPLYE), as well as interestingness of an issue report (INT). The variability in interestingness (INTE) shows the same statistically significant and positive relation to the risk of post-release defects, that we have observed earlier in our case study on ECLIPSE.

> *Overall, we observe that the introduction of the third dimension (communication dynamics) leads to a four to five times increase of explanatory power in the model compared to the previous two dimensions (+2.51%). This increase is statistically significant. Similar to our findings in* ECLIPSE*, discussion flow and interestingness are strongly connected to post-release defects.*

Model M4 introduces the fourth dimension of social interaction metrics used in our study: workflow activity. In contrast to our findings on ECLIPSE 3.0, our results show a statistically significant, strongly negative relation between the amount of workflow activities (WA) and post-release defects in FIREFOX 3.0. Upon manual inspection of the workflow activities in FIREFOX, we found a large amount of supporting workflow activities, such as the addition of attachments, modification or addition of supporting meta-information, such as version information, keywords, quality assurance contacts, or testing and debugging information. In contrast, we found a relatively strong emphasis on actual process activities such as the re-assignment of the issue to a different developer, and other status changes that commonly relate to bug tossing [Guo2011] in ECLIPSE.

> *Overall, the introduction of workflow activity metrics increases the explanatory power of the model by 1.48%. This increase is statistically significant. Our findings indicate a strong negative relation between workflow activities and post release defects.*

With *Model M5*, we introduce the dimension of social structures by adding our measures of experience for participants. We observe a strong increase in explanatory power of 16.6% over the previous models, upon introduction of experience metrics. We observed a similar large increase in our case study on the ECLIPSE project. Figure 6.6 presents a plot of odds-ratios of the presence of experienced participants in the discussion on post-release defects. Similarly to our case study on ECLIPSE, we observe a small number of distinct spikes, indicating particular experienced discussion participants (denoted by a particular index on the x-axis rather than their actual names for privacy reasons), whose presence in a discussion of an issue report stands in a strong relationship (the higher the spike in y-direction, the higher the odds-ratios) with post-release defects.

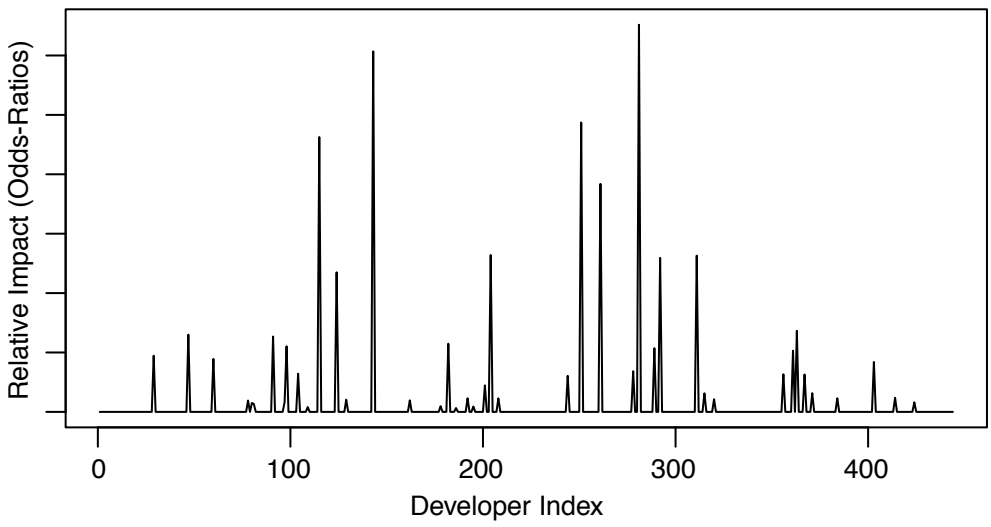


Figure 6.6: Odds ratios for experience metrics in model M5, each index represents one distinct level (developer) of the factor variables.

Overall, the introduction of experience metrics increases the explanatory power by 16.60%. A type II ANOVA test confirms that this increase is statistically significant. Similar to our findings in ECLIPSE*, we observe a relatively large connection between post-release defects and the presence of particular contributors in discussions.*

6.5.4 *Additional Versions of Firefox*

We repeated the same hierarchical modeling approach for two additional releases of FIREFOX. The final models are presented in Table 6.10 and Table 6.11. Analysis of variance inflation factors led to the exclusion of NMSG and NDEVS in FIREFOX 1.5, and the exclusion of NMSG, NDEVS, as well as NPATCH in FIREFOX 2.0. As a result, NPATCH was only available for FIREFOX 1.5, where is was not deemed statistically significant, and NDEVS was not available for any release

$\log(Y_i)$	MB		M1		M2		M3		M4		M5	
CHURN	**6.437**	*	**6.000**	*	**6.114**	*	**5.250**	*	**4.761**	*	**4.618**	*
NSOURCE			**1.162**	*	**1.104**	∘	**1.159**	•	**1.161**	•	**1.189**	•
NSCOM			**1.320**	*	**1.780**	*	**2.407**	*	**2.378**	*	1.279	
PATCHS			0.568		0.477		0.676		0.731		0.360	
NLINK			**1.073**	∘	**1.348**	*	**1.267**	*	**1.265**	*	1.046	
NPART					**0.474**	*	0.682		**0.641**	+	**1.938**	+
NUSERS					**1.150**	∘	1.055		1.068		**0.662**	*
SNACENT					2.931		7.403		**14.568**	+	0.140	
REPLY							**1.054**	∘	**1.050**	∘	1.048	
REPLYE							**0.397**	+	**0.384**	∘	**0.055**	*
DLEN							**0.856**	*	**0.879**	•	1.040	
DLENE							**0.075**	*	**0.081**	*	**0.046**	*
INT							**0.907**	∘	**0.888**	∘	**0.716**	*
INTE							**4.148**	*	**2.212**	∘	**4.004**	∘
WA									0.966		0.945	
WAE									**4.190**	•	**4.672**	∘
Chi-Sq	1269.4(	*	1365.9(	*	1432.3(	*	1583.2(	*	1593.3(	*	2760.3(	*
Dev.Expl	14.75%		15.87%		16.64%		18.40%		18.51%		32.07%	
Delta Chisq			96.46		66.42		150.95		10.05		1167	

⋆ p<0.001, • p<0.01, ∘ p<0.05, + p <0.1

Table 6.10: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics for FIREFOX 1.5. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

of FIREFOX.

Overall, we observe a similar increase in explanatory power for each dimension, as seen in the detailed case study on FIREFOX 3.0. This is especially true for the large increase in explanatory power, when including experience metrics in model M5. We find the following metrics consistent across all three releases of FIREFOX: CHURN, NSOURCE, NSCOM, PATCHS, NLINK, SNACENT, INTE and WAE are statistically significant and connected with an increased risk of post-release defects. REPLYE and WA are statistically significant and connected with a decreased risk of post-release defects. On the whole, we observe a larger set of consistent metrics for FIREFOX, than in the case of ECLIPSE.

$\log(Y_i)$	MB		M1		M2		M3		M4		M5	
CHURN	**4.336**	*	**4.328**	*	**4.450**	*	**2.754**	*	**2.643**	*	**4.162**	*
NSOURCE			0.973		0.966		1.018		1.019		1.073	
NSCOM			1.121		1.107		**1.548**	•	**1.548**	•	1.041	
PATCHS			2.219		2.155		3.258		3.425		7.294	
NLINK			**1.084**	◦	**1.185**	*	**1.208**	*	**1.198**	*	0.967	
NPART					1.051		**0.592**	+	**0.610**	+	0.780	
NUSERS					**0.607**	*	**0.678**	*	**0.685**	*	**0.651**	◦
SNACENT					1.320		**74.664**	◦	**81.532**	◦	3.180	
REPLY							**1.036**	+	**1.034**	+	**1.089**	◦
REPLYE							**0.446**	+	**0.493**	+	0.399	
DLEN							**0.829**	*	**0.860**	*	**0.866**	◦
DLENE							**0.134**	*	**0.198**	*	0.848	
INT							**1.302**	*	**1.254**	*	**1.411**	•
INTE							**10.095**	*	**4.948**	*	**5.228**	•
WA									**0.912**	+	**0.871**	+
WAE									**4.380**	•	2.261	
ChiSq	**1292.6**	*	1298.9		**1347.9**	*	**1605.9**	*	**1621.3**	*	**3942.5**	*
Dev.Epl.	12.67%		12.74%		13.22%		15.75%		15.90%		39.15%	
DeltachiSq			29.74		33.66		125.75		73.82		829.32	

⋆ p<0.001, • p<0.01, ◦ p<0.05, + p <0.1

Table 6.11: Hierarchical analysis of logistic regression models along the four dimensions of social interaction metrics for FIREFOX 2.0. For every stepwise model, we report odds ratios for each regression coefficient, as well as a goodness of fit metric (χ^2), the percentage of variation in the data each model explains (Dev. Expl) and the difference in goodness of fit compared to the previous hierarchical modelling step ($\Delta\chi^2$).

6.6 *Comparison of Case Study Results*

In this section, we present a summary and comparison of our findings from both case studies. For the purpose of comparing the regression variables of each model, we cannot directly compare the odds ratios, as discussed in Section 6.3. Instead we follow a standard approach in statistics, which has been successfully used in previous research [Mockus2009]: to gain a comparable notion of the relative effect of each regression variable in each model, we first compute the mean values of each regression variable across the whole population from which the regression model was built. We use these mean values as an input into the linear equations connected with model $M4$ in each case study. This equation has the form $Y = \beta_0 + \beta_1 v_1 \ldots \beta_n v_n$, with β_i the

	Delta Y					
Variable	**Eclipse 3.0**	**Eclipse 3.1**	**Eclipse 3.2**	**Firefox 1.5**	**Firefox 2.0**	**Firefox 3.0**
CHURN	0.17984	0.15167	0.20010	0.22208	0.13972	0.22852
DLEN	-0.01965	-0.06054	-0.01836	-0.00456	-0.02757	0.02017
DLENE	0.00831	0.07840	-0.02389	-0.02616	-0.07111	0.07934
INT	-0.02897	0.00949	0.02386	-0.02585	0.03184	-0.09742
INTE	0.00031	0.00272	0.02155	0.01951	0.04016	0.02838
NDEVS	-0.13279	NA	-0.03286	NA	NA	NA
NLINK	0.01686	-0.10109	0.00224	0.02695	0.02154	0.04899
NPART	0.21818	NA	NA	NA	NA	0.01139
NPATCH	-0.00627	0.00036	NA	0.00072	NA	NA
NSCOM	0.00775	-0.01751	-0.01571	0.05092	0.02221	0.01969
NSOURCE	0.05033	0.01809	0.04377	0.02422	0.00254	0.00283
NTRACE	-0.00311	-0.00562	-0.00435	NA	NA	NA
NUSERS	-0.02833	0.06777	0.00546	0.00090	-0.02641	0.04512
PATCHS	0.00691	0.00024	0.00023	0.00348	0.00018	0.00068
REPLY	-0.00252	-0.01563	0.00513	0.00934	0.00588	-0.00231
REPLYE	-0.04589	0.02008	0.03073	-0.02301	-0.01810	-0.09136
SNACENT	NA	-0.04969	NA	-0.11774	0.06465	0.02924
TRACES	-0.00166	0.01027	0.01706	NA	NA	NA
WA	0.05967	0.08872	-0.05113	-0.00126	-0.01686	-0.12480
WAE	0.02832	0.02687	-0.02282	0.02728	0.02912	0.01106

Table 6.12: Relative effect (Delta Y) of a 20% increase of a predictor variable on post-release defect probability. Entries marked as NA are variables excluded from the model through VIF analysis.

regression coefficients reported earlier, and v_i the mean values for each regression variable. For each regression variable, we then increase that one variable by 20%, keeping all other regression variables constant, and obtain a value Y_E. The difference $\Delta_Y = Y_E - Y$ describes the relative effect of each regression variable on post-release defect probability within its respective range. Table 6.12 presents the results of this analysis for each project and release. To increase readability, we have ordered the regression variables for each model according to Δ_Y in decreasing order.

To further enable a simpler comparison of the effect of regression variables on post-release defect probability across releases and projects, we summarize our observations from Table 6.12 in Table 6.13. Regression variables that are statistically significant at $p < 0.005$ are marked in bold font face. When a regression variable increased the probability of post-release defects we put the value “POS”, whenever a regression variable decreased the probability of post-release

	Mozilla			Eclipse		
	v1.5	**v2.0**	**v3.0**	**v3.0**	**v3.1**	**v3.2**
CHURN	**POS**	**POS**	**POS**	**POS**	**POS**	**POS**
NSOURCE	**POS**	POS	POS	**POS**	**POS**	**POS**
NSCOM	**POS**	**POS**	**POS**	POS	**NEG**	**NEG**
NPATCH	NEG			**NEG**	POS	
PATCHS	POS	POS	**POS**	**POS**	POS	POS
NTRACE				NEG	NEG	NEG
TRACES				NEG	**POS**	**POS**
NLINK	**POS**	**POS**	**POS**	**POS**	**NEG**	**POS**
NUSERS	POS	**NEG**	**POS**	NEG	**POS**	**POS**
NDEVS				NEG		**NEG**
SNACENT	**POS**	**POS**	POS		**NEG**	
DLEN	**NEG**	**NEG**	POS	**NEG**	**NEG**	**NEG**
DLENE	**NEG**	**NEG**	POS	POS	**POS**	**NEG**
REPLY	**POS**	**POS**	**NEG**	NEG	**NEG**	**POS**
REPLYE	**NEG**	**NEG**	**NEG**	**NEG**	**POS**	**POS**
INT	**NEG**	**POS**	**NEG**	**NEG**	**POS**	**POS**
INTE	**POS**	**POS**	**POS**	POS	**POS**	**POS**
WA	NEG	**NEG**	**NEG**	**POS**	**POS**	**NEG**
WAE	**POS**	**POS**	POS	**POS**	**POS**	**NEG**

Table 6.13: Summary of effect direction of each regression variable on the probability of post-release defects. Statistically significant regression variables at $p < 0.005$ are marked in bold font.

defects, we put the value "NEG". Variables that were removed by VIF analysis or were not available in a project (such as TRACES and NTRACE in FIREFOX), are left blank.

Overall, we note that only a few variables show a statistically significant connection to post-release defects across all releases and projects. In particular, these variables are: CHURN, NLINK, REPLYE, and INT. We also observe a number of regression variables, that are deemed statistically significant for one project but not for the other. In particular these variables are: NSCOM in FIREFOX, SNACENT in ECLIPSE, DLEN in ECLIPSE, INTE in FIREFOX, WA in ECLIPSE, and WAE in ECLIPSE.

Apart from statistical significance, the more interesting observation is the direction of effect, each variable has on the probability of post-release defects. We observe, that CHURN, NSOURCE, PATCHS and INTE are consistently connected with an increased risk of post-release defects across

all releases of both projects. For both, code churn (CHURN) and number of files modified by patches (PATCHS), these findings are in line with previous work [Hassan2009; Nagappan2007]. However, the relationships between number of source code examples and post-release defects, as well as interestingness entropy and post-release defects are neither obvious, nor intuitive, and open research opportunities for future work. Overall, we would like to note, that these variables (in addition to code churn) might be valuable for the use in defect prediction across releases and projects.

Furthermore, we observe a number of regression variables that are consistent within one project. In particular, these variables are: NSCOM, NLINK, SNACENT, REPLYE, WA and WAE for FIREFOX, as well as NDEVS, DLEN, and SCNACENT for ECLIPSE. These variables might still be valuable for the use in defect prediction within the same project, across multiple releases of the software.

In addition to the particular relationships between single regression variables and the probability of post-release defects, we observe a number of consistent trends for our overall prediction models. In particular, the four dimensions of social interaction metrics are able to improve the explanatory power of models between 2.17 (FIREFOX 1.5) and 3.08 (FIREFOX 2.0) times the power of the baseline models, built on code churn. Second, each dimension adds a statistically significant amount of explanatory power of the previous dimensions (except for dimension two, social structures in the case studies of FIREFOX 2.0 and ECLIPSE 3.0).

6.6.1 *Discussion of Entropy Measures*

In order to interpret the observations for variables, which capture the variability of particular metrics across different observations (messages, discussions, files), we discuss each such variable in more detail in the remainder of this section.

For each assessment of entropy metrics, we first split the original dataset from which we built the regression models used in our case studies into two parts: one part contains all the

Metric	Eclipse 3.0	Eclipse 3.1	Eclipse 3.2	Firefox 1.5	Firefox 2.0	Firefox 3.0
DLENE	—	$\mu_1 > \mu_2$	$\mu_1 > \mu_2$	$\mu_1 > \mu_2$	$\mu_1 > \mu_2$	$\mu_1 > \mu_2$
REPLYE	$\mu_1 > \mu_2$	—	$\mu_1 > \mu_2$	—	$\mu_1 > \mu_2$	$\mu_1 > \mu_2$
INTE	—	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$
WAE	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$	$\mu_1 < \mu_2$

Legend: μ_1 denotes the mean entropy for files with no post-release defects
μ_2 denotes the mean entropy for files with post-release defects
— denotes Mann-Whitney test did not reject H_0 at $p < 0.005$

Table 6.14: Summary of Entropy Measure Analysis for ECLIPSE and FIREFOX.

files that had at least one post-release defect in the period of 6 months after release; the other part contains those that had none. For each part, we then collect the measurements of entropy and compare both distributions of measurements using an unpaired two-sided non-parametric statistical test, called the Mann-Whitney-U test, which is more robust against outliers than classical test, such as Student's t-test and does not rely on the assumption that the underlying data has normal distribution [Fay2010]. The results of this analysis are summarized in Table 6.14 and discussed in the following.

A. Discussion of Entropy in Discussion Length (DLENE)

For both projects, we consistently observe a higher mean entropy for discussion length in files that had no post-release defects, than in files that had post-release defects. Except for `ECLIPSE 3.0`, the Mann-Whitney test confirms that this difference is statistically significant at $p < 0.005$.

> *For* `ECLIPSE` *and* `FIREFOX`*, files that have no post-release defects exhibit more consistency in the length of the discussions on issue reports, connected to those files. Hence, we conclude that a larger variability in wordiness of discussions is connected to an increased risk of post-release defects.*

B. Discussion of Entropy in Reply Time (REPLYE)

For both projects, we consistently observe a statistically significant connection between the variability in reply times between discussion messages, and post-release defects. For `FIREFOX 2.0` and `FIREFOX 3.0`, as well as for `ECLIPSE 3.0` and `ECLIPSE 3.2`, we find that the mean value of reply time entropy is higher for files that had no post-release defects, than for files that had post-release defects. Except for `FIREFOX 1.5` and `ECLIPSE 3.1`, the Mann-Whitney test, confirms that the differences in both projects are statistically different at $p < 0.005$.

Since a lower measure of entropy value is connected to a greater spread of values, we conjecture that for both projects outliers in reply time, i.e., a significant delay in the flow of the discussion, is connected to a larger risk of post-release defects. Our findings thus indicate that inconsistencies in information flow stand in relation to post-release defects.

C. Discussion of Entropy in Interestingness (INTE)

For both projects, we observe that the mean entropy for interestingness is lower in files that had no post-release defects, than in files that had post-release defects. Except for `ECLIPSE 3.0`, the Mann-Whitney test confirms that the difference is statistically significant at $p < 0.005$.

Files that have no post-release defects exhibit a larger variability of interestingness across all the issue reports connected to that file. At the same time, files that have post-release defects show a more consistent interestingness across all issue reports connected to that file. Hence, we conclude that a larger variability in interestingness is connected to a decreased risk of post-release defects.

D. Discussion of Entropy in Workflow Activity (WAE)

We observed a statistically significant connection between workflow activity entropy and post-release defects for both, FIREFOX and ECLIPSE. Similarly, we find for both projects, a lower mean entropy for workflow activity in files that had no post-release defects, than in files that had post-release defects. A Mann-Whitney test confirms that the difference between both distributions is statistically significant at $p < 0.005$ in all cases.

> *Files that have no post-release defects exhibit a larger variability of workflow activity across all the issue reports connected to that file. At the same time, files that have post-release defects show a more consistent workflow activity across all issue reports connected to that file. We hence conjecture that workflows involving a greater variety of steps are connected to an increased risk of post-release defects.*

6.7 *Enhancing Traditional Models with Social Information*

Through the two case studies presented in Section 6.4 and Section 6.5, we have demonstrated that statistical models based on social interaction metrics yield an increase in explanatory power of up to 16.36% (ECLIPSE) to 21.86% (FIREFOX) over the baseline model using code churn.

In this part of our analysis we want to investigate whether social information metrics can augment existing, top-performing defect prediction models that are based on an extensive set of source-code and file metrics. To perform this comparison, we use a publicly available defect prediction dataset, which was prepared by the University of Saarland [Zimmermann2007]. As Bird et al. note [Bird2009b], this dataset is extensively documented and has been widely used in research.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-4.7228	0.2475	-19.08	0.0000
log(1 + pre)	1.1766	0.0890	13.22	0.0000
log(1 + MLOC_max)	-0.2049	0.0606	-3.38	0.0007
log(1 + PAR_max)	0.4081	0.1324	3.08	0.0021
log(1 + PAR_sum)	-0.1599	0.0888	-1.80	0.0716
log(1 + TLOC)	0.8058	0.0936	8.60	0.0000

Table 6.15: Baseline Model M

Among other information, this data set contains a variety of source-code and file-level metrics for files of different Eclipse releases. We are specifically interested in the latest release contained in this dataset, Eclipse 3.0, as we measured the social interaction metrics presented in this study during the same time period. Both datasets contain a different set of metrics for the same source code files: Zimmermann et al.'s dataset contains source code metrics, whereas our dataset contains social interaction metrics. We will use a combination of both datasets for the remainder of this section, to study the effects of combining both sources of information.

We first re-create the original code-metrics based model created by Zimmermann et al. [Zimmermann2007]. The original statistical model M is presented in Table 6.15. This model contains the following regression variables: `pre` denotes the amount of defects associated with the file in the past 6 months before release. `MLOC_max` measures the maximum amount of non-blank, non-comment lines of source code inside method bodies. `PAR` measures the number of parameters inside method signatures, both as a maximum across all methods in a file, and as a sum of all methods. `TLOC` denotes the total lines of code in a file, including blank lines and comments.

We assess it using the same criteria as the models we derived from our hierarchical analysis. Our results show that the model has an explanatory power of $\chi^2 = 889.48$ (17.04% of deviance explained) and all regression variables are statistically significant at $p < 0.1$.

Next, we create an extended model M' by adding the set of social interaction metrics that we found statistically significant in our hierarchical analysis of `ECLIPSE` 3.0, to model M. This extended model is presented in Table 6.16. We observe that the addition of social interaction

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-5.2783	0.2860	-18.46	0.0000
log(1 + pre)	0.9341	0.0987	9.46	0.0000
log(1 + NSOURCE)	0.5128	0.0657	7.81	0.0000
log(1 + NPATCH)	-1.5056	0.7009	-2.15	0.0317
log(1 + PATCHS)	2.0683	0.9862	2.10	0.0360
log(1 + NLINK)	0.5146	0.1175	4.38	0.0000
log(1 + REPLYE)	-2.0122	0.5515	-3.65	0.0003
log(1 + WA)	0.3919	0.0796	4.92	0.0000
log(1 + MLOC_max)	-0.1839	0.0614	-3.00	0.0027
log(1 + PAR_max)	0.3795	0.1354	2.80	0.0051
log(1 + PAR_sum)	-0.1743	0.0908	-1.92	0.0550
log(1 + TLOC)	0.7939	0.0950	8.36	0.0000

Table 6.16: Augmented Model M'

metrics increases the explanatory power of the new model M' by $\chi^2 = 716.55$ to a total of $\chi^2 = 1606.03$. This corresponds to an increase of 13.73% percent of additional deviance explained, to a total of 30.77%. An ANOVA test confirms that the observed increase over the baseline model M is statistically significant at $p < 0.001$. To compare, the best performing model based on source code metrics was reported by Shihab et al., with a total of 21.2% of deviance explained [Shihab2010a]. By adding social interaction metrics, we are able to greatly outperform this model, demonstrating the additional value of social information for defect prediction models.

The large increase in explanatory power of the augmented model demonstrates that social interaction metrics are valuable for complementing traditional prediction models based on source-code based product and process metrics.

6.8 Related Work

In the following, we discuss related research from the two major research areas of defect prediction and social analyses of software development. Several researchers have previously investigated the use of data captured from version control systems and bug databases for defect prediction. Basili et al. [Basili1996] established and promoted the usefulness of object-oriented code metrics for predicting the defect density of code. Ohlsson and Ahlberg were among the first researchers to use code-oriented metrics to predict failure prone modules of a software [Ohlsson1996].

Extensive work by Nagappan et al. [Nagappan2005; Nagappan2007] has investigated the value of code and churn metrics to predict defects in large-scale commercial systems. Schroeter et al. showed that module dependencies, which are already available at design time can be used to predict software defects [Schroter2006].

Hassan demonstrates in a large case study that prediction models based on change complexity outperform traditional churn based prediction models [Hassan2009]. Zimmermann et al. use social network measures on dependency graphs to predict defects [Zimmermann2008a].

In contrast to previous research, the work presented in this study does not focus on formulating accurate prediction models. We rather focus on using statistical models and the insights about relationships between variables that can be gained from studying these models, to investigate the relationships between social interactions and software quality. As such, we use prediction models as an explorative tool in the same vein of work done by Mockus et al. [Mockus; Mockus2009].

Shihab et al. [Shihab2010a] carried out an analysis of the variation inflation factors of the source-code based process and product metrics, initially reported for the ECLIPSE project by Zimmermann et al. [Zimmermann2007]. Our work extends this study by adding social interaction metrics to the defect prediction model presented in the study by Shihab et al.

The work by Wolf et al. [Wolf2009] presents a case study on the use of social network analysis measures obtained from inter-developer communication in the IBM Jazz repository to predict build failures. Similar use of socio-technical network measures to predict software defects has been carried out by Pinzger et al. [Pinzger2008] and Meneely et al. [Meneely2008].

A related study was conducted by Bacchelli et al. [Bacchelli2010a] that investigates the possible use of code popularity metrics obtained from email communication among developers for defect prediction. Our work differs from these studies, as we use a variety of measures of social information to study relationships between these measures and the strength of their associations rather than performing actual predictions.

A recent study by Jeong et al. uses workflow activity recorded in the issue tracking system of the ECLIPSE project to improve issue assignment. Together with the work of Guo et al. [Guo2010; Guo2011] as well as Shihab et al. [Shihab2010b], empirical evidence on the possibly negative effects of workflow activity served as an intuition for including workflow activity metrics for defect modeling in our work.

6.9 *Threats to Validity*

We discuss the limitations of our study and the applicability of the results derived through our approach. For this purpose we discuss our work along four types of validity [Yin2009]: construct validity, internal validity, external validity and reliability.

6.9.1 *Construct Validity*

The assessment of construct validity aims at evaluating the meaningfulness of measurements and whether they quantify what we want them to. The conjecture of our work is that the

social interactions between developers and users, which surround the development process of a software system has an impact on the quality of the final software product. In order to capture social interaction, we use issue tracking systems as a repository containing records of such interaction. We focus on issue tracking systems over less formal and structured repositories, such as mailing lists, as issue tracking system are well understood and contain a wealth of historic data surrounding the evolution and maintenance of a software system. Furthermore, they contain meta-data that allows us to reliably establish traceability links back to the source of a software system.

One big assumption of our work is that the source code and issue repositories capture all the data, which might generally not be the case. However, the quality and extend of data recorded in open-source projects is likely very high, as development teams in open-source are often distributed across different countries and timezones, and thus heavily depend on the completeness of data, for the success of their collaborative software development efforts.

At the core of interaction in a software community are discussions, such as the discussion on the reported issues recorded in issue tracking systems like BUGZILLA. With respect to issue report discussions, we defined a set of metrics along four different conceptual dimensions. Our metrics of the first dimension, discussion contents are based on a previous survey of developers [Bettenburg2008c] and focus on quantifying the presence of those information items that developers regarded as most helpful when working with issue reports (i.e., fixing a software defect). We capture these information items through an automated approach, which has been evaluated in previous research [Bettenburg2008c; Bettenburg2008b].

Our metrics of the second dimension, social structures, assume that the actors in the issue tracking system can take on one of two roles: developer or user. We follow previous work in the area [Jeong2009; Sliwerski2005] and define a developer as an actor in the system who has been assigned to working on at least one issue in the past. However, there might be actors present in the system, who are developers, but have never been assigned to fixing a bug. Furthermore, we capture the expertise of participants through determining the past amount of contributed

messages to discussions, as past research has demonstrated that issues reported by experts have a higher likelihood of being successfully closed [Guo2010]. Our approach is limited by recording only the top three experts of each discussion. We chose a threshold of three, to keep the artificial inflation of our models through the introduction of "dummy" variables low. In order to capture the structural properties of a discussion, we use a metric from social network analysis, closeness-centrality [Wasserman1994], which captures the extent, to which a participant talks to every other participants. As closeness-centrality is a per participant metric, we aggregate over all participants by the use of normalized entropy. A healthy discussion, in which every participant interacts with every other participant would thus be characterized by maximum entropy. Our approach could explore the use of other metrics from the area of social network analysis, or aggregate per participant metrics differently.

Our metrics of the third dimension, communication dynamics, focus on the quantitative measurements of a discussion, such as the number of exchanged messages, their length, time and interestingness. Our approach approximates interestingness as the degree of exposure to actors in the bug tracking system. For this purpose we use the notification list, on which participants can sign up to be notified when the information on an issue changes. By design of the issue tracking system, every participant in a discussion is automatically put on the notification list. Furthermore, issues might be interesting to actors, but actors might decline to sign up on the notification list, e.g., to avoid additional email traffic.

Our metrics of the fourth dimension, workflow, focus on the development activities that are recorded by the issue tracking system [Cubranic2003]. These activities follow preset development workflows, from creation of an issue to closure. However, we cannot directly observe any workflow activities beyond those that are recorded in the issue tracking system. While some issue tracking systems, such as FIREFOX record an extensive set of workflow activities, our manual inspection of the data revealed that the workflow in the ECLIPSE bug tracking system is mostly concerned with activities related to issue management.

For all dimensions, our approach is limited by capturing only a small subset of possible

metrics. Using the same line of work, we could extend the set of metrics of social interaction to accommodate and explore further hypotheses of relations between social interaction and post-release defects, beyond those presented in this work.

In order to judge whether the models M1 to M5 obtained through our hierarchical modeling approach describe valid relationships, rather than random observations in the data, we have carried out tests to judge the possible over-fitting of every statistical model (the test is described in detail in Section 4.3). The only instance of over-fitting we have observed was caused by the inclusion of experience metrics CON1-CON3 in model M5 for both projects, and across all releases. As a result, we have removed the experience metric from our hierarchical modeling approach and performed comparisons of models across releases and projects, as well as the discussions of results based on the most complete model, which does not suffer from over-fitting: Model M4.

6.9.2 *Internal Validity*

The assessment of internal validity deals with the concern that there may be other plausible hypotheses that explain our findings. Furthermore, can we show that there is a cause and effect relation between the processes captured through our metrics of social interaction, and post-release defects?

Our approach uses regression models to put a set of regression variables (our metrics of social interaction) into a relation with a dependent variable (post-release defects). The effect of each variable is described through odds ratios. However, odds ratios describe only the magnitude and direction a unit change of the independent variable will have on the dependent variable, while keeping all other variables constant. Second, we can only observe correlation through statistical models, not causation. As such, all observations that we report on, even though they describe statistically significant connections, denote that we observed a co-occurrence of certain properties. In order to investigate possible causal effects, we would need to carry out a root-cause

analysis along each variable.

For each observation we attempt to give an intuitive rationale for that particular connection. Where applicable, we carried out a manual inspection of our collected data with a specific observation in mind. However, there may be plausible rival hypotheses to explain our observations. Our work builds the basis for future investigations, by identifying a number of statistically significant connections between social interactions and software quality. At the same time, some of our observations confirm findings of previous studies: code churn being closely tied to post-release defects [Nagappan2007] and the spread of changes being connected to post-release defects [Hassan2009].

6.9.3 *External Validity*

The assessment of external validity evaluates to which extent generalization from the results of our study are possible. We have performed two case studies on large-scale open-source software systems with different domains: `ECLIPSE` is an integrated development environment, `FIREFOX` is a web-browser. In addition, we have studied multiple releases of each software system to further reduce threats to external validity. However, since processes and practices differ greatly between open-source development and commercial development, our observations might not generalize to industrial settings. We believe, that the approach described in this paper, together with the provided resources, can be readily adopted to other projects, as long as these projects provide records that allow to establish traceability links between the source code and issue tracking system.

6.9.4 *Reliability*

The assessment of reliability of our study refers to the degree to which someone analyzing the data presented in this work would reach the same results or conclusions. We believe that the

reliability of our study is very high. Our data is derived from publicly available source control and issue tracking systems. In addition, we provide all underlying source code, datasets and analysis scripts in a replication package at http://sailhome.cs.queensu.ca/replication/social-interactions/.

6.10 Conclusions

Summary

In this chapter we investigated the connection between developer communication, expressed through a set social interaction metrics (which we measured from discussions surrounding issue reports mined from issue tracking systems) and software quality. Our results establish and illustrate these connections through the regression variables of statistical models. To allow for a detailed study of the strength of these connections, we partitioned measurements about developer communication and social interactions between developers during development activities into four separate dimensions (discussion contents, social structures, communication dynamics, and workflow). For each of these dimensions, our results express the relative strength of the relationship between that dimension and software quality as odds ratios in logarithmic space, as well as through a separate statistical analysis. Our results make key contributions to two areas of empirical software engineering: defect prediction and understanding.

Defect Prediction

Our results not only confirm the consistent relationship between code churn and increased risk of post-release defects presented by previous research in this area, but also establish a set of

metrics that might be equally valuable for defect prediction. These new metrics center around different aspects of how developers interact and communicate during software development activities.

In particular we found that the number of source code snippets discussed by developers and users (NSOURCE) are related to an increased risk of post-release defects. This finding is consistent across all releases and projects studied. Further research is needed to investigate the intricacies of this connection, e.g., are code examples needed for meaningful discussions of more complex or error-prone code changes.

Other metrics that were consistently connected to an increased risk of post-release defects include the spread of changes proposed through patches (PATCHS), the variability of interestingness (INTE), as well as variability of workflow (WAE). At the same time, we presented metrics that were consistent across several releases of the same project, but not across different projects. These include the overall number of stack traces discussed (NTRACES), information flow effectivity (SNACENT), and overall workflow activities (WA).

Our findings demonstrate that a combination of both, source-metrics based models and social metrics based models yields a higher predictive power than either of the models on its own. Our results thus support the findings of previous research in the area (ref Chapter 2) that software engineering is a highly social process and that social aspects of how developers interact during development activities plays an important role in the quality of a software.

Developer Communication and Software Quality

The other focus of our research is on understanding the relationships between developer communication and software quality. Through our results we observed that models based on social interaction metrics not only explain a similar amount of software defects as traditional models, which use source-code based product and process metrics, do, but that we can use social information to complement traditional models to obtain higher explanatory power than each

model taken on its own. In other words, measurements derived from observations about how developers communicate and interact with each other from a sociological point of view are not merely a mirror of what we can measure from a technical point of view on the software, but are a disjunct set of observations that bring key value to creating a larger, holistic picture when trying to understand software quality.

6.10.1 *Relevancy to this Thesis*

In this chapter, we investigated the first part of our thesis hypothesis, that *communication between developers impacts software quality*. The findings of the multi-case study presented in this chapter strongly support this hypothesis. Among the main findings presented in this chapter, we demonstrated that not only *what* developers talk about, but also, *how* they discuss development activities stands in a strong relationship to software quality.

We feel that our findings confirm the value of information about developer communication for the software engineering research community. We believe that our work makes a strong case for establishing social metrics as a promising direction to explore in future research in empirical software engineering, for both the prediction and the understanding quality communities.

7

The Impact of Communication on the Evolution of the Software

In recent years, many companies have realized that collaboration with a thriving user or developer community is a major factor in creating innovative technology driven by market demand. As a result, businesses have sought ways to stimulate source code contributions from developers outside their corporate walls, and integrate external developers into their development process. The communication between volunteer developers from the community and core developers of the software is a central part of how external source code contributions are managed and integrated into the software. In this chapter, we investigate developer communication surrounding source code contributions. In particular, we investigate this relationship through an empirical study on the contribution management of two major, successful, open source software ecosystems (Android and the Linux kernel). We base our analysis on a conceptual model of contribution management that we derived from a total of seven major open-source software systems. This model defines five phases of contribution management and we describe the role of developer communication in each of these phases. Through a case study, we show that delayed communication can valuable cause code contributions to be abandoned.

7.1 Introduction

Over the past decade, open-source as a business model has gained in popularity and studies have documented benefits and successes of developing commercial software under an open-soure model [Krishnamurthy2005]. Companies like RedHat, IBM or Oracle, realize that collaboration with a thriving user and development community around a software product can increase market share and spawn innovative products [Hecker1999]. One of the major benefits of an open-source business model is user-driven innovation. As opposed to traditional, in-house development models, an open-source model gives the users of a software system the ability to actively participate in the development of the product, and contribute their own time and effort on aspects of the product that they find most important.

At the core of innovation are contributions, such as source code, bug fixes, feature requests, tutorials, artwork, or reviews, received from the community surrounding a software product. These contributions add to the software product in many ways, such as adding new functionality, fixing software defects, or completing and translating documentation.

However, involving a community in the development process may require significant changes to development processes and communication between in-house and community developers, as both external and internal contributions need to be accommodated at the same time. This leads to a number of challenges that might not be obvious at first. For instance, a lack of transparency in communication between external and core developers may lead to duplicate implementations of the same functionality [Bettenburg2013b].

Past literature provides only limited insight into how contribution management is carried out in practice. In the context of this research question, we aim to learn about contribution management from documented processes and practices of 7 major, successful open-source projects, and abstract our observations into a step-by-step model that covers essential steps in the management process. In particular, how are code contributions communicated from the community

to the core developers responsible for integrating these contributions into the software product?

In a systematic approach, we first derive a conceptual model of contribution management that spans five steps, from initial inception of a contribution to the final integration of the contribution into the product and delivery to the end-users. We then proceed to study the relationships between developer communication surrounding code contributions to the evolution of the software along the five steps of the conceptual model. Through a case study on contribution management of two large, successful open-source software systems, the Linux kernel, and Google Android, we demonstrate that timely communication plays a central role in the management of code contributions from community developers.

7.1.1 Contributions

We identify the main contributions of the work presented in this chapter as:

First, A conceptual model of the contribution management process with the goals to: a) methodologically derive an abstraction of the contribution management of successful, large open source projects from scattered project documentation, and b) to provide a common basis for terminology and practices of contribution management.

Second, descriptive statistics and recommendations to practitioners based on case studies of two real world instances of contribution management processes and a quantitative assessment of their success.

Third, an investigation of developer communication surrounding code contributions, demonstrating that communication plays a key role in the successful evolution of a software.

7.1.2 *Organization of this Chapter*

The rest of this chapter is organized as follows. We present a conceptual model for contribution management that we derived from seven major for-profit and not-for-profit open source systems in Section 2. In Section 3, we discuss how ANDROID and LINUX realize contribution management in practice through a case study that follows our conceptual model of contribution management. In Section 3, we present our case study on LINUX and ANDROID, which follows along the five phases of contribution management established through the conceptual model presented in Section 2. In Section 4, we discuss potential threats to the validity of our study and outline the precautions we have taken to balance these threats. We proceed with identifying and discussing related work in Section 5, followed by our concluding remarks in Section 6.

7.2 *Contribution Management Model*

In order to establish a common ground for studying contribution management processes with respect to terminology and concepts, we first derive a conceptual model of contribution management. In the same way that architectural models create abstractions of actual instances of software implementations and give researchers a common ground for studies, our conceptual model aims at being a starting point for establishing common terminology when talking about contribution management.

The model of contribution management presented in this section was derived through a systematic study of publicly accessible records of processes and practices in this chapter's subject systems, ANDROID and LINUX, as well as five additional popular for-profit and not-for-profit open source software systems from different domains. The seven software systems that we used to derive our conceptual model from are summarized in Table 7.1. These seven projects were selected among contemporary, prominent and important open-source projects, so we could

understand how mature projects do contribution management. All projects have enough public exposure (press coverage, availability of documentation and discussion lists surrounding the development and contribution processes) to perform a systematic analysis of the surrounding documentation to derive a meaningful picture [Strauss1990; Charmaz2006]. Furthermore, we selected mature projects that are well-established in the open-source field. We derived system size through public source code statistics provided by Ohloh[1], where available, and from press releases of the software systems otherwise.

We derive the conceptual model from publicly available documents about the seven subject systems, by following an approach known as "Grounded Theory" [Glaser1967; Strauss1990]. Grounded Theory aims at enabling researchers to perform systematic collection and analysis of qualitative textual data with the goal to develop a well-grounded theory. The process starts with a maximally diverse collection of documents and follows three separate steps. The first step, called "Open Coding", consists of reading and re-reading all documents and identifying, naming and categorizing phenomena observed in the qualitative data. In the second step, called "Axial Coding", the analyst relates the categories derived in the first step to each other with the aim of fitting concepts into a basic frame of generic relationships. Through the third step, called "Selective Coding", the analyst distills the core concepts and abstractions of the observations in the qualitative data.

We start our derivation process on contribution management practices in Open Source Software (OSS) by first collecting and analyzing the publicly available documentation for each project, press releases, white papers, community mailing lists, and websites that document each project, as well as research literature in the area of open source software engineering. We started our abstraction of a common model by understanding the workflow that a contribution undergoes in each project. Some projects, such as ANDROID provide very detailed documentation[2], whereas workflow in other projects is documented less explicitly by the members of

[1] http://www.ohloh.net
[2] http://source.android.com/source/life-of-a-patch.html

Project	Led By	Domain	Model	System Size
OPENSOLARIS111	Company (Oracle)	OS Environment	Commercial product features cherry-picked from open source code.	10M LOC
ECLIPSE 3.6	Foundation (Eclipse Foundation) funded through membership fees from individuals and companies	IDE and Framework	Commercial product builds on top of open source product.	17M LOC
MySQL 5.5	Company (Oracle)	Database System	Support, certification and monthly updates for enterprises.	1.3M LOC
ANDROID 2.3	Company (Google)	Embedded Device Platform	Operates separately sold hardware.	73M LOC
APACHE 2	Foundation (Apache Software Foundation) funded through donations	Web Server	Not-for-profit	2M LOC
LINUX Kernel	Individual (Linus Torvalds)	OS Kernel	Not-for-profit	14M LOC
FEDORA 14	Foundation (Fedora Project) sponsored by Company (Redhat), Community Governed	OS Environment	Commercial product features cherry-picked from open source code.	204M LOC

Table 7.1: Overview of for-profit (OPENSOLARIS, ECLIPSE, MySQL, ANDROID) and not-for-profit (FEDORA, APACHE, LINUX) open source projects studied to extract a conceptual model for contribution management.

the project themselves and needs to be recovered from more anecdotical sources. For example, in OPENSOLARIS, we recovered workflow information from multiple sources, in particular, the community Wiki and development mailing lists. Following the example of previous research in the area [Asundi2007; Rigby2008], we then looked for commonalities across all projects and finally divided the contribution management processes into individual steps that are common across all projects.

An inherent threat to the validity of such a derivative process is that we can claim neither completeness, nor absolute correctness of the derived theory. However, our aim was to derive a first abstraction, which we leveraged from the records and descriptions of actual implementations of contribution management processes in practice. This abstraction serves on the one hand as a starting point for discussing contribution management throughout our study on a scientific basis, and on the other hand we hope that future research will pick up and incrementally refine this abstraction with what is known in the field, similar to conceptual models of software architecture.

7.2.1 *Conceptual Model*

Overall, the derived conceptual model consists of five phases that a contribution undergoes before it can be integrated into the next release of a software product and be delivered to the community. In the following subsections, we discuss each phase in the order of labels presented in Figure 7.1, and illustrate each phase with concrete examples from the software systems that were analyzed. We observe that in each phase, communication between the community and core developer plays a central role.

Phase 1: Conception

Similar to classical software development, prospective contributors with an idea for a new feature or bug fix often seek early feedback and support from the rest of the community, as well as the internal development teams, to work out their ideas into concrete designs. Such discussions usually take place in public project mailing lists (e.g., `OPENSOLARIS`, `APACHE`), issue tracking systems (e.g., `ECLIPSE`), Wikis (`FEDORA`), and/or special purpose discussion forums (`MySQL`).

The outcome of the conception phase is either a concrete design (usually after multiple rounds

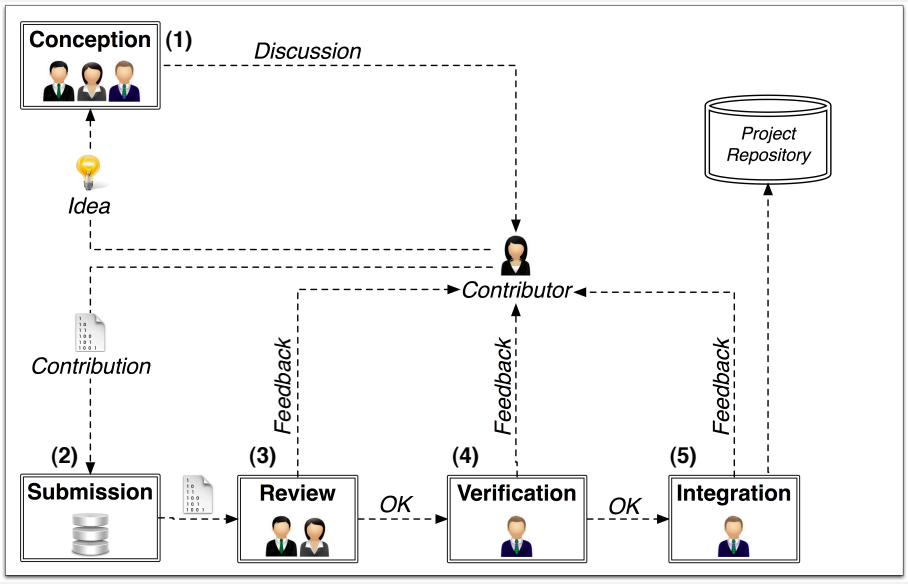


Figure 7.1: Conceptual Model of the Collaboration Management Process.

of feedback), or a rejection of the proposed contribution, if the idea does not align with the project's or the community's goals. The conception phase is not mandatory – in some projects contributors skip this phase altogether and start with a concrete source code submission that was designed individually.

Phase 2: Submission

Once the design for a contribution has been fleshed out in source code, a contributor submits the contribution through the submission channels provided by the project. Since many of these submission come from external contributors (community members), intellectual property infringements are a substantial concern [German2009]. All seven projects that we studied acknowledge this risk and have established policies for their submission processes that guarantee traceability of the submission to the original author.

For example, `ECLIPSE`, `FEDORA`, `MySQL` and `APACHE` completely disallow contributions through mailing lists, as the identity of the sender can not be verified. Instead, they require

a submission to be carried out formally by opening a new record in their issue tracking systems.

Phase 3: Contribution Review

After a submission has been submitted for consideration, it will ideally reach senior members of the project (even though there is no guarantee that this is always the case, as our data on the LINUX system demonstrates). All seven projects require a formal peer review to be carried out for every submitted contribution. Contribution review has the following three goals.

1. Assure Quality. Senior developers may catch early on obvious issues of the contribution and possible source code errors, and give the contributor a chance to address these problems.

2. Determine Fit. As community members are often unaware of internal development guidelines and policies, the primary goal of the review phase is to determine the overall fit of the contribution for the project and ensure that contributions meet the established quality standards.

3. Sanitize Code. Reviewers check contributions for programming guidelines and standards, inappropriate language, or revise comments intended for internal viewing only. As part of the sanitization process, developers may also review the contribution for use of third-party technology, such as usage of external libraries whose licensing might not align with the project [German2009], as for example practiced in the ECLIPSE project.

The review phase has three potential outcomes: a contribution is accepted as-is, a contribution needs to be reworked, or a contribution is rejected. In case there are concerns with the contribution, reviewers are encouraged to communicate with the author of the contribution, in order to give feedback on the reasons of the rejection. The contribution author is then expected to either address any concerns raised and re-submit, or abandon his contribution.

Phase 4: Verification

After a contribution passes the review phase (often after multiple iterations), senior members of the project team or a project lead need to verify whether a contribution is complete and free of software errors (e.g., making sure the contribution passes all regression tests). The verifier is

the person who has the final say on whether a contribution gets accepted or not. If any problems arise during the verification phase, the verifier(s) can give feedback to the original contributor, who can then resubmit an updated revision of the contribution (back to Step 2).

Common reasons for contributions being rejected during the verification phase include software errors, incompatibilities with the most recent version of the project repository (e.g., they target an out-dated branch of the software that is no longer actively developed or maintained), or strategic decisions [Wnuk2009]. The verifier has the ultimate say and can reject contributions that received positive reviews in the previous phase if he does not see a fit for the contribution in the long term direction of the project. For example, the contribution correctly implements a certain feature, yet an alternate version for the same feature is already planned to be copied from another upstream project that also implemented the same feature independently.

Since verification is a tedious step, some projects try to automate or outsource this process. For example, in FEDORA and ECLIPSE testing during the verification phase is crowd-sourced through nightly builds (daily updated builds that are not meant for public release), which contain the latest contributions for testing by the community. In addition, build and testing infrastructures and tools such as JENKINS [3] are becoming increasingly advanced and enable (semi-)automated verification of contributions in the context of the existing software.

Phase 5: Integration and Delivery

If a contribution has passed the peer review, is technically sound, and has been verified, it enters the integration phase. The goal of this phase is to integrate one or more contributions that have passed review and verification, into the software product, and to ultimately deliver it to the community. Integration of a contribution is often challenging, as the contributed code may stand in conflict with the source code of other contributions, as well as internal changes. If integration fails, contributors are often required to adapt their contributions to remove conflicts and work together with the most recent revision of the development repository.

[3] http://jenkins-ci.org/

Strategies for integration range from an immediate merge into the publicly available source code repositories, to delivery of contributions as part of a release schedule, such as daily builds, or official releases [Duvall2007]. For special contributions, such as critical bug fixes, or high-impact security issues, strategies for a fast-tracked integration of contributions are valuable for reducing ill-effects.

Intellectual Property (IP) Management

Interestingly, across all projects that we studied, we found only a single instance of a particular IP management process - in the case of ECLIPSE, a closed-access tool called IPZILLA [4] is used to internally check for IP issues with contributions from third parties. However, we found a number of "best-practices" documented across different projects that support the management of Intellectual Property through keeping track of contributor's identities by the following means:

1. **Login Credentials.** For ECLIPSE, FEDORA, MySQL and ANDROID, users need a valid and active registration in the online system used to facilitate the contribution process (BUGZILLA, GERRIT). Users are required to provide their name, email address (and in some cases a postal address) to successfully register in the system. Contributions are then associated to their unique user id or user handle in the system. During the registration mechanism of the GERRIT tool in ANDROID, a user has to explicitly agree to, and sign a Contributor License Agreement with Google that covers the transfer of IP rights for any submitted contribution.

2. **Formal Registration.** For APACHE[5] and OPENSOLARIS[6], users are required to print a special form called the Contributor License Agreement (CLA), sign it and mail, email, or fax it to a central authority to become registered as a code contributor. Only contributions from successfully registered individuals are considered.

[4] https://dev.eclipse.org/ipzilla

[5] www.apache.org/licenses/icla.txt The Apache Software Foundation Individual Contributor License Agreement (CLA)

[6] www.opensolaris.org/os/sun_contributor_agreement The Sun Contributor Agreement (SCA)

3. **Developer Certificate of Origin.** For LINUX, contributions are submitted by electronic mail. Instead of contributor submitting an individual contributor licensing agreement (CLA) such as we have seen in 1. and 2., emails are required to contain a "Sign-Off" field identifying the contributor through a name and email address pair. The version control system (GIT) tool includes a special command line option "–signoff" to automatically sign submissions to the code repository with the credentials that a user has provided in the tool configuration.

4. **Firewalling.** A common practice across all seven projects is to further restrict the rights for modification of the main source code repository to a small set of "committers". Individuals earn commit rights by demonstrating technical skill through continued contribution to the project as non-committers (in this case another committer needs to sponsor their contributions), and by gaining social reputation, i.e., the more well-known and trusted individuals are by their peers, the more likely they will be given permissions to change the code directly [Bird2007].

5. **Internal Review.** Through an interview with ECLIPSE developers we learned that the ECLIPSE Foundation carries out internal reviews of submission to check for third-party license conformity. We could not find any such practice documented for any of the other six projects, but suspect that similar practices are in place even though they are not explicitly described.

Relationship to other Socio-Technical Participation Models in OSS

Since the beginning of the Open-Source Software development phenomenon, researchers have sought to understand how and why developers join an open-source project, and what their properties of participation in the project's development process are.

For example, the *private-collective* model presented by von Hippel and von Krogh [Hippel2003] conjectured how external users contribute to software projects, in order to solve their

own problems on the one hand, and technical problems that are shared among the community surrounding the software on the other hand. In their study of major open source projects, von Hippel and von Krogh find that such external contributors freely reveal their intellectual property without commercial interests, such as private returns from selling the software.

In particular, for the APACHE webserver, and the FETCHMAIL email utility, von Hippel and von Krogh describe how the private-collective paradigm of providing contributions to a software for a public good impacts the organization and governance of traditional (commercial) development projects [Hippel2003]. In particular, von Hippel and von Krogh conjecture the existence of governing entities (team leaders, core-developers) and project resources, which our study confirms and describes in detail in the next section. For example, we observe in both of our case study subjects the existence of leadership roles as proposed in the private-collective model. However, the setup of these leadership roles differs significantly across both our case study subjects: while the LINUX project is governed in a hierarchical, pyramid-like fashion of increasing level of authority, with a "benevolent dictator" at the top and a hierarchy of meritocratically chosen lieutenants below, we find that the ANDROID project is governed by a two-tier "board of directors" approach, where the authority is in the hand of Google employees.

While the private-collective model studies open-source participation and contribution from a user incentive-level, the participation model presented by Sethanandha et al. focuses on understanding how individual pieces of code in the form of patches are contributed by external users and are handled by the project [Sethanandha2010]. In contrast to the work by Sethandha et al., the conceptual model presented in this chapter covers a more complete picture of the contribution process, beyond the submission and handling of patches. In particular, our conceptual model integrates conception, verification and integration phases, which are crucial parts of the overall management of contributions from users.

7.3 Case Study on Android and Linux

In the following, we present our case study on the role of communication during contribution management in two open-source software projects. We begin by describing our data collection process, followed by a brief definition of metrics used in our case study.

We then present a quantitative assessment of the role of communication in the contribution management processes of ANDROID and the LINUX kernel. In particular, our case study follows the five phases of the conceptual model of contribution management presented in the previous section.

7.3.1 Data Collection

To investigate the impact of communication surrounding code contributions on the evolution of a software, we study contribution management in two well-established open-source projects: the LINUX kernel and ANDROID. This choice is motivated in particular by a) LINUX being a prime example of a thriving, long-lived project, which spearheaded the open source movement and b) ANDROID being a for-profit open source project, which is backed by a very successful major software company (Google). Even though ANDROID is publicly available for free, the underlying business interest is still for-profit, as the mobile platform allows for sale of search, advertisements, and mobile apps, and hence the product is treated like most commercial software. The selection of both projects was based on four key characteristics:

1. Both projects are especially successful in the open source software domain.

2. Both projects receive a large quantity of contributions from their communities. These contributions help evolve the product, perform corrective and perfective maintenance, as well as a multitude of extra services (e.g., creation of graphics, art, tutorials, or translations), and extend the product halo.

3. Both projects are system software, so we can observe how they compare.

4. One project is for-profit, the other not-for-profit, which might effect the communication surrounding the contribution management processes.

For both projects, we carried out a quantitative analysis by analyzing the available source code repositories, mailing lists discussion repositories, and a qualitative analysis through inspection of publicly available web documents, and project documentation. For LINUX, we also took experience reports and previous research [Mockus2000a; Rigby2008] into account. For the ANDROID system, we relied on Google's publicly-available documentation of processes and practices, as well as empirical knowledge derived from analysis of the source code and available data from the GERRIT code review system.

Mining Communication Data Surrounding Contribution Management

In LINUX, contributions are submitted through the LINUX kernel mailing lists. There is one global mailing list, and multiple specialized mailing lists for the different kernel subsystems. For our case study on LINUX, we first downloaded and mined the email repositories of 131 LINUX kernel mailing lists between 2005 to 2009. We then used the MAILBOXMINER tool (ref. Chapter 3) to extract all communication data between developers and the community from the LINUX email repositories.

To enable the study of code contributions, we use the approach presented in Chapter 4 to extract contributions (which are submitted in LINUX in the form of patches) from the communication data. Based on this data, we extracted metrics on both the mailing lists (number of participants, frequency, volume, etc.), and the actual contributions themselves (size, files modified, complexity, etc.).

To obtain information on which contributions are accepted, we developed a heuristic, which splits up a contribution per file that is being changed (this part of a contribution is called a "patch chunk"), extracts a list of files that are being modified by the contribution, removes noise

such as whitespace, and then calculates a hash string in the form of a "checksum" of the relative file path and the changed code lines [Jiang2013]. This is done to uniquely identify patches, as formatting changes widely across email (i.e., whitespace, line endings, word and line wrap). We follow the same approach to produce unique hashes for the accepted patches in the main LINUX GIT version control system. We then match contributions extracted from emails to contributions that were accepted into the main LINUX repository by comparing their hashes, and also taking into account the chronological aspects of email and commits (i.e., we do not link a patch that was accepted into the main repository if an email that contains a contribution with a matching hash was sent at a later date). Once this mapping from individual parts of contributions that found their way into the main LINUX source code repository to contributions that were submitted to the kernel mailing lists is established, we abstract back up from patch chunks (that describe parts of contributions) to user contributions.

We manually evaluated the performance of our linking approach on a sample of 3,000 email discussions that contain the identifiers of actual GIT commits, in which the discussions' patches were accepted. We measure the performance of our approach by means of "recall" (i.e., how many of those that we know were linked in reality are also being linked by our approach). Upon manual inspection of a random stratified sampling of 100 emails (both linked and not linked) across all kernel mailing lists, we found that our approach had a precision of 100% and a recall of 74.89% on this sample. According to statistical sampling theory, this provides us with a 10% confidence interval around our result at a 95% confidence level (i.e. we are 95% sure that the performance achieved with our approach lies between 90% and 100% precision). Both measures, precision and recall, provide us with confidence that the metrics we calculated on LINUX contributions are sufficiently accurate to obtain meaningful descriptions of the overall population of contributions sent over the LINUX kernel mailing lists.

Through these analysis steps we were able to recover a) what are the contributions that are communicated over the LINUX mailing lists, b) who are the actors, c) which parts of the contributions get eventually accepted and d) all the surrounding meta-data such as dates, times,

discussions on the contributions, as well as contribution size. These four parts of information form the main body of data for our study.

In ANDROID, contributions from the community are communicated to the core developers entirely through a dedicated server application, called GERRIT. GERRIT has a front-end user interface, which consists of a (client-side) web application running on Javascript[7]. Use of GERRIT is mandatory for any user who wants to submit a contribution to the ANDROID project. The user designs and carries out code changes in a local copy of the project repository. When done, the user needs to submit the entirety of the changes done as a delta to the main project repository. GERRIT, which sits on top of the distributed version control system GIT, monitors these submission requests, intercepts the request, puts it on hold and automatically opens a new code review task. Developers then discuss the submitted source code contribution through a message system within GERRIT. Only when this code review has been approved and verified by a senior developer (as described in Section 2), does the contribution get sent on to the main project source code repository for integration.

The code review itself contains a large amount of meta-information surrounding the contribution management process, such as discussions on the proposed change, discussions on the source code, actors involved in the contribution and its review, as well as their votes in favor or against the contribution. In particular, the GERRIT system is designed to allow multiple people to vote and sign off changes in the distributed development system, as well as to inspect and comment on the source code itself.

According to an interview with Shawn Pearce, the lead developer of GERRIT, on FLOSS weekly[8], the main goal of GERRIT is to provide a more formal and integrated contribution management system compared to an email-based approach found in projects like the LINUX kernel.

[7] https://https://android-review.googlesource.com/
[8] http://google-opensource.blogspot.ca/2010/05/shawn-pearce-on-floss-weekly.html

Since the GERRIT front-end like many other Google Web Toolkit (GWT) applications runs entirely as a client-side javascript program in the browser, classical approaches of mining the HTML pages for data (often referred to as "web-scraping") do not work. As a workaround, we created a custom mining tool that directly interfaces with the GERRIT server's REST services. The publicly-available source code of the GERRIT system provides us with the necessary APIs to make REST calls to the server, retrieve data in the form of JSON objects and serialize these into Java Bean classes. We then copy the information contained in the Java Beans into a local database for further analysis. One advantage of this approach is that the Java Beans contain much richer data than what is already presented in the web-front end. In particular associations of entries with unique user ids, date precision and internal linking of artifacts are some of the highlights that greatly help our later analyses.

Overall, we obtained a snapshot of all the publicly visible contributions to the ANDROID project that were recorded through the GERRIT system from the start of the project, until July 2010. Our dataset contains 6,326 contributions from 739 active accounts that were contributed over the course of 16 months, starting from the initial open-source release of the ANDROID project.

The GERRIT dataset contains:

a) the source code of the individual contributions

b) multiple versions of each contribution in the cases they needed to be revised

c) discussions surrounding the contributions' source code (on a line-level)

d) discussions surrounding the contribution management process including review and integration

e) all votes that led to a decision on each

f) meta-data such as dates, times, unique user ids, dependencies and other traceability links to related artifacts and repositories contribution granularity)

A manual inspection of a random sample of 100 entries suggests that the data GERRIT provides is both complete and clean, i.e., the "raw" data has all meta-data attached that we needed for this study, and no further cleaning or pre-processing steps were necessary.

7.3.2 *Definitions*

In the following, we refer to the state of LINUX during the year 2005 (the earliest data after the most recent switch of their contribution management process) as LINUX 05 and to the state of LINUX during the year 2009 (the same time period for which we collected ANDROID data) as LINUX 09. We split the LINUX dataset into two parts to make comparisons between LINUX and ANDROID fairer. In particular, LINUX 05 describes the first year of LINUX' current contribution management process, and similarly, the ANDROID dataset describes ANDROID's first year of contribution management. For both datasets, we collected data up to the following of the reported year, and for counting returning contributors in particular have followed through to that point (e.g., for ANDROID, we report on data until January, but looked until July to see if those commits' developers sent anything later on).

- **Feedback Time.** The time from submission of the code contribution until a first response on the contribution is sent to the original author of the discussion. We use this measure as a response variable in our analyses.

- **Review Time.** The time from submission of the code contribution until a final decision on whether the contribution is accepted into the software or rejects. We use this measure as a response variable in our analyses.

- **Message Length.** The number of words in the message that is attached to the contribution during submission. Our motivation to include this measure is that lengthy messages might indicate contribution that require additional documentation or explanation.

- **File Spread.** The number of files are added or modified by the contribution. Our motivation to include this measure is that the more files are touched by a contribution, the higher the chance that multiple people need to coordinate the evaluation of the contribution.

- **Contribution Size.** The numbers of lines of code the contribution contains. Our motivation to include this measure is that the larger the contribution, the more time it might take to evaluate the contribution.

7.3.3 *Lifespan of Community Developers*

We measure the activity of community developers as the time span between the date of their first activity (contribution, message, review, or comment) until the date of their last recorded activity. We leave out those community members for which only a single activity was recorded and ignore any possible hiatus between periods of activity (i.e., a contributor's period of activity begins with the date of submission of that contributor's first submission, and ends with the date of the last recorded submission by that contributor).

We find that the median period of activity of community members in ANDROID is 65 days. In LINUX 05, community developers have a median period of activity of 24 days, whereas in LINUX 09, the median activity is 57 days.

To get an idea of the potential impact of the lifetime of community developers on contribution management, we carried out a manual inspection of all ANDROID contributions that were submitted by one-time contributors. We found that 19.51% of these contributions were abandoned, as reviewers took longer than two months (=60 days) to initiate a discussion on the contribution with the original author, who was no longer available. The remaining 80% of abandoned contributions were due to a variety of reasons, such as:

- the contributor implemented functionality that was already implemented in-house, but in the non-public development branch. As the development branch is not publicly available,

the contributor was not aware of this duplication. (48.7%)

- the contributor implemented functionality that was already implemented by another contributor at the same time. (7.3%)
- the contribution conflicted with the master source code repository beyond feasible repair. (4.8%)
- the contribution was submitted to the wrong subsystem. (7.3%)
- the contribution was deemed incomplete or of too low quality. (36.6%)

Community developers stick around only for a limited period of time after they submit a contribution. We conjecture that communication speed, in particular fast feedback to community developers, is a key factor for the successful evolution of the software through code contributions. When first feedback is given to developers after they are no longer available, contributions may end up in an unfinished state and are ultimately abandoned, thus wasting precious resources of both, contributors, and reviewers, that could have been directed elsewhere.

7.3.4 *Phase 1: Conception*

Both ANDROID and LINUX provide mailing lists for the conception and discussion of new ideas. The main difference with respect to contribution management is the traceability of a contribution from initial conception to final implementation. As ANDROID uses different technologies for conception (mailing lists) and all other phases (the GERRIT tool), the conception phase is separated from the remaining contribution management process, resulting in weak traceability.

LINUX on the other hand uses the same email discussion for both, conception, as well as review, thus enabling practitioners to follow a contribution from the initial idea to the final encoding in source code. The practices in LINUX have evolved historically: LINUX has been under

development for almost 20 years and even though processes and technologies were adapted along the way (i.e., using different Version Control Systems to manage the project's source code repositories), the social process has mostly stayed the same.

RQ1: How does communication about a contribution during the conception phase impact the subsequent management of the contribution?

Linux'05				
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	39489.5812	5283.5089	7.47	0.0000
msg_length	32.7179	147.0835	0.22	0.8240
contrib_spread	-70.9020	121.3079	-0.58	0.5589
contrib_size	-2.8500	3.2028	-0.89	0.3736
nr_pre_contrib_messages	-564.8889	276.7088	-2.04	0.0412
Linux'09				
	Estimate	Std. Error	t value	Pr(>|t|)
(**Intercept**)	197033.9021	7423.9017	26.54	0.0000
msg_length	176.2311	195.5178	0.90	0.3674
contrib_spread	-184.4842	268.2205	-0.69	0.4916
contrib_size	0.6717	1.5619	0.43	0.6672
nr_pre_contrib_messages	-4571.4193	1528.2989	-2.99	0.0028

Table 7.2: Modeling Time until First Response on an Contribution (in seconds), by length of the message (msg_length), number of files added or modified by the contribution (contrib_spread), the size of the contribution in number of lines of code (contrib_size), and number of messages exchanged on the contribution topic before the submission of the code contribution (nr_pre_contrib_messages). Predictors marked in bold font are statistically significant at $p < 0.005$

To answer this research question, we first model response time, i.e, the amount of time passed between the initial submission of the contribution, until a first feedback on the contribution is given to the contribution author. As we have no communication data for Android during the conception phase, we build statistical models for the Linux kernel only. We present our models in Table 7.2.

We observe that the amount of discussion preceding the submission of the contribution is related to a faster time until a reviewer first responds to the author of the contribution with feedback in both `LINUX 05` and `LINUX 09`. Messages with no previous discussion took on

Linux'05			
	Chi-Square	d.f.	P
msg_length	46.50	1.00	0.00
contrib_size	1.22	1.00	0.27
contrib_spread	158.10	1.00	0.00
nr_pre_contrib_messages	15.77	1.00	0.00
Linux'09			
	Chi-Square	d.f.	P
msg_length	137.17	1.00	0.00
contrib_size	27.93	1.00	0.00
contrib_spread	33.93	1.00	0.00
nr_pre_contrib_messages	1.10	1.00	0.29

Table 7.3: Modeling Contribution Acceptance Probability, by length of the message (msg_length), number of files added or modified by the contribution (contrib_spread), the size of the contribution in number of lines of code (contrib_size), and number of messages exchanged on the contribution topic before the submission of the code contribution (nr_pre_contrib_messages).

average 1.383 times longer to receive first feedback (65.2 hours in 2009) than messages that had a preceding discussion (47.1 hours in 2009).

In addition, we create a statistical model to investigate whether communication during the conception phase has an impact on whether the contribution will be accepted into the software or not. We present the results of an analysis of variance test (ANOVA) to determine the statistical significance and explanatory power of each predictor variable in the corresponding statistical model in Table 7.3. Overall, we observe a shift from the number of files added or modified by a contribution as the strongest predictor for acceptance in LINUX 05 to the length of the message that describes the initial contribution in LINUX 09. While the amount of discussion preceding a submission has small explanatory power in LINUX 05, we cannot find a statistically significant relation (at $p < 0.05$) between either the amount of discussion preceding submission, or the time to receive a first feedback on the contribution and the acceptance of the contribution into the software.

7.3.5 *Phase 2: Submission*

Contributions to ANDROID are submitted to the project's master version control system through a special purpose tool provided by the project. During the submission process a new peer review task is automatically opened in the GERRIT system that blocks the contribution from being delivered into the master version control system until a full review of the contribution has been carried out.

In contrast, LINUX uses mailing lists for the submission of contributions. These mailing lists act as a repository that is detached from the version control system. Contributions are propagated to the version control system later on manually, once a contribution has been accepted in the peer review process of the submission.

RQ2: How does feedback on a contribution impact subsequent management of the contribution?

Based on our earlier observations on the lifespan of community developers, we believe that timely feedback to contributors is a key factor to keep the author of the contribution engaged and available for feedback.

For ANDROID, we find that the average time until initial feedback is given *decreased* by a factor of 3.55, from 221.91 hours (approximately nine days) for contributions submitted in March, 2009 to 62.37 hours (approximately two and a half days) for contributions submitted in March, 2010. In contrast, we find that for LINUX, the average time until initial feedback *increased* by a factor of 1.37, from 37.63 hours (one and a half days) in 2005 to 51.40 hours (about two days) in 2009.

While the increase of feedback delay in LINUX can be explained by an increase in per-contributor submissions on the one hand, as well as a reduction of available reviewers per contribution, we found no such evidence for ANDROID.

However, a plot of the raw data of response times for ANDROID presented in Figure 7.2

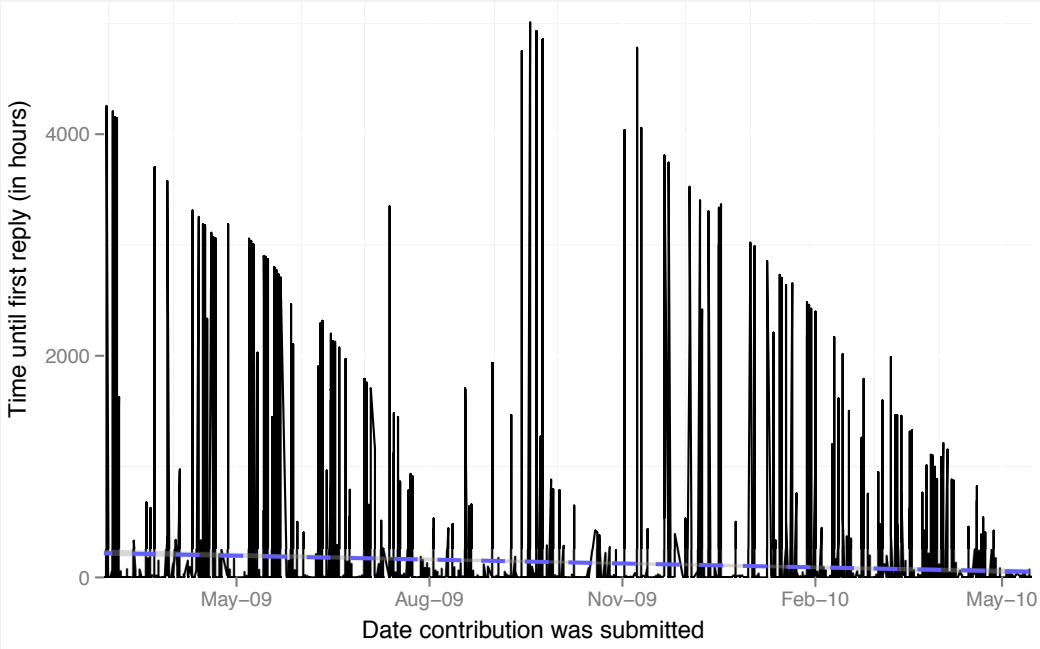


Figure 7.2: The time until a contributor is given first feedback on a submission in ANDROID.

reveals a series of triangular patterns. These patterns indicate the early struggles of the ANDROID project to give timely feedback: before June 2010, the established practice was to batch-process all contributions in the system around the time of a major release (October 2009 for ANDROID Eclair 2.0 and May 2010 for ANDROID Froyo (2.2)).

The recorded feedback times lie on an almost perfect slope that meets the x-Axis at the major release dates. This practice led to a variety of problems, such as abandoned contributions due to a lack of interactivity between senior members who judged contributions and the contributors (who had already moved on and were no longer actively engaged in the project).

In June 2010, Jean-Baptiste Queru, one of the lead developers of ANDROID announced a radical change of the review process, as documented on the ANDROID development blog [9]

*"We're now responding to [*ANDROID *] platform contributions faster, with most*

[9] http://android-developers.blogspot.com/2010/06/froyo-code-drop.html

changes currently getting looked at within a few business days of being uploaded, and few changes staying inactive for more than a few weeks at a time. We're trying to review early and review often. [...] I hope that the speedy process will lead to more interactivity during the code reviews."

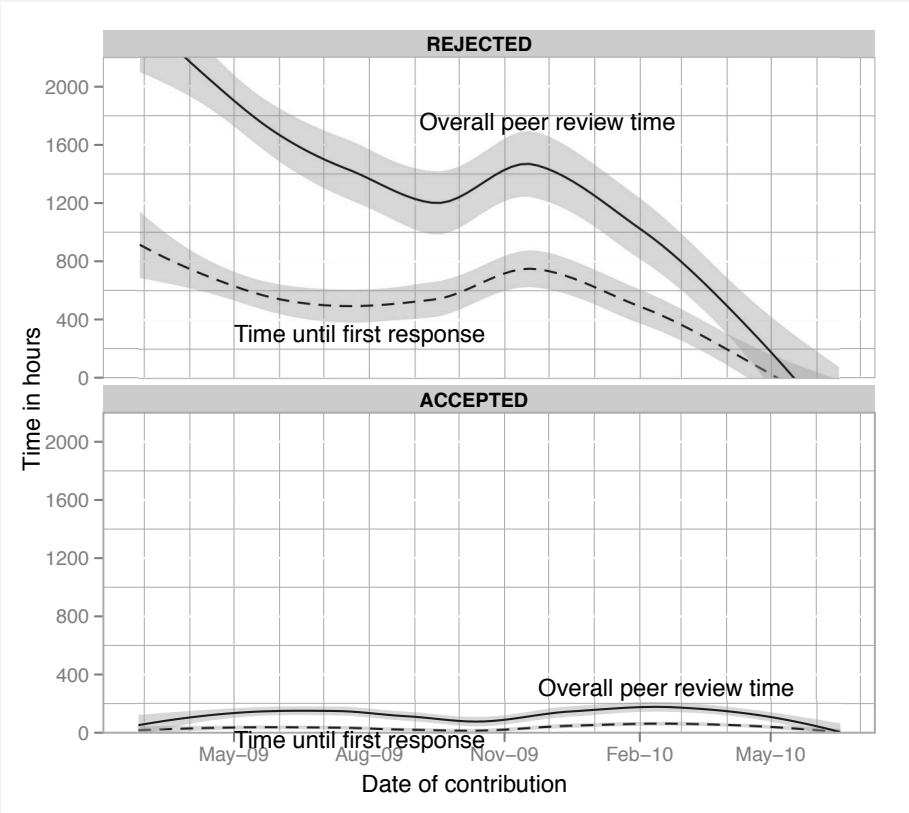


Figure 7.3: Feedback and Review Time in ANDROID by final decision outcome.

Upon further analysis of the ANDROID dataset, we observed that distributions of the review times are biased towards rejection (see Figure 7.3). A common practice in ANDROID is to perform clean-up of the GERRIT system right before a software release. As part of that clean-up phase, contributions that have been open for a long time and that are not actively pursued, e.g., a reviewer has been waiting for the contributor to submit an updated version of the patch for an extended period of time, are closed with a rejected status.

Even though we observe a three-fold increase in response time in LINUX, the results of our analysis show that the impact on overall review time is relatively small (ca. 7%). This finding suggests that, while the self-managed appointment of reviewers in LINUX is suited for absorbing a large increase in submission volume without causing an overall increase in review times, initial feedback suffers. One possible explanation could be that a large increase in contribution volume also increases the amount of email messages community members receive, and thus more choice when picking contributions for review.

The ANDROID project switched from a periodical batch-processing of contributions around the time of major releases to a process that allows for early feedback and frequent reviews. This change in process was purposely done to increase interactivity between senior developers and contributors during the review phase of the contribution management process. Through the case study on LINUX, we documented the need of contribution management processes to account for increases in submission volume and community size to avoid delays. This is especially important to avoid losing valuable contributions due to contributors having only a short timespan in which they are active and available for feedback.

7.3.6 *Phase 3: Review*

Pending Review Notification

In ANDROID, each contribution triggers an automated notification to the project leaders and the appointed reviewers of the subsystem for which the contribution was submitted. This process is automated and handled by the GERRIT system. Reviewers can then access the new contribution through the web interface and make their thoughts known through either attaching a message to the general discussion of a contribution, or commenting directly on specific lines in the contribution's source code. Either action triggers in return an automated notification to the original author of the contribution, as well as the other reviewers.

Peer review in LINUX is organized less formally. In addition to a project-wide mailing list for

overall discussion, there exist many subsystem-specific mailing lists. Contributors are encouraged to submit their contributions through the corresponding mailing list of the subsystem that their contribution targets. However, functionally this is not much different from queues in GERRIT. Every community member subscribed to the subsystem mailing list sees new entries and can act on them as she sees fit. If a community member voluntarily decides to carry out a peer review, they can do so freely and at any time by sending their review as a reply to the corresponding email discussion.

Judgement of contributions

In ANDROID, contributions are judged formally through positive and negative votes, cast by reviewers and verifiers. Only contributions that have at least one vote for acceptance (a +2 vote) can move on to verification. In particular, reviewers in ANDROID can cast a +1 vote to indicate that the contribution should be accepted, and a -1 vote to indicate that the contribution should be rejected. Senior members, i.e., verifiers, then assess the votes of the reviewers and decide on acceptance of the contribution (they cast a +2 vote), or rejection (they cast a -2 vote).

The judgement of contributions in LINUX is less formal. Contributions are either *abandoned*, if the community showed not enough interest (with respect to follow up emails on the original submission); *rejected*, if a project leader decides that there were too many concerns raised by the community; *revised*, if there were only minor concerns that could be corrected easily; or *accepted* as is. In contrast to ANDROID, acceptance of a contribution in LINUX is implicit: a contributor knows whether his contribution was accepted, when the maintainer of a subsystem accepts the contribution in his own copy of the master version control system. In case of revision, updated versions of the contribution are commonly submitted to the same email thread.

RQ3: How long does it take to complete a review of the contribution?

In LINUX, the average time needed to complete the review phase *increased* by 7.2% from 183.80 hours (approx. seven and a half days) in 2005, to 197.90 hours (approx. eight days) in 2009.

In contrast, we find that for ANDROID, the average time to complete the review phase *decreased significantly* from 522.20 hours (about 21 days) in March 2009 to 80.34 hours (about 3 days) in March 2010. As we discuss in the following, this decrease is largely due to two effects: first, the practice of performing clean-up before a release of ANDROID, and second, active efforts in decreasing feedback delays.

In addition, we investigate the relation between overall review time and outcome through kernel-density analysis [Rosenblatt1956]. The plots derived from this analysis are presented in Figure 7.4.

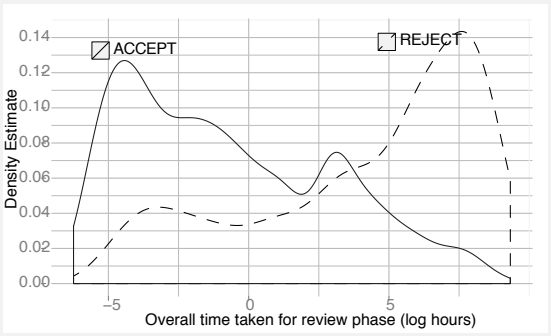
Our kernel density plots are estimates of the probability density functions of the random variables connected with acceptance or rejection of contributions. The actual values of the y-axis in these cases depict the probability of the random variable attaining the value at the corresponding point on the x-axis. Within this context, kernel density plots should be viewed as very precise histograms. The big advantage is that histograms can appear greatly different depending on the number of bins an analyst specifies and easily over or under-sample the data at hand, while kernel density estimates automatically derive an optimal bandwidth.

Our observations show that reviewers in ANDROID are fast in deciding whether to accept a contribution, but take much more time to reject a submission (Figure 7.4a). Most considerably, we observe the opposite for LINUX. From 2005 to 2009, decisions on whether to reject a contribution take up increasingly less time, and decisions about accepting a contribution take up increasingly more time (Figure 7.4b and Figure 7.4c).

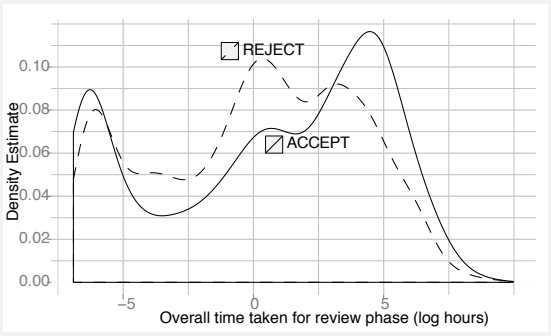
In LINUX, we observe that contributions are rejected quickly, yet a decision for acceptance takes considerably more time. In ANDROID, we observe that contributions are accepted quickly but a final decision towards rejection takes much longer.

A possible explanation for the observed increase in overall review time, as well as feedback time from LINUX 05 to LINUX 09, might be a decreased ratio of reviewers to contributions, as well as an increased submission volume per contributor.

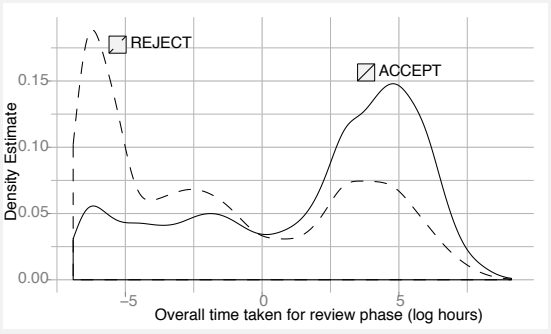
To study this hypothesis, we categorized community members into two classes, contributors



(a) ANDROID



(b) LINUX **05**



(c) LINUX **09**

Figure 7.4: The overall time taken to reach a decision differs significantly across projects.

and reviewers. We consider a community member as a contributor, if our data contains at least one submission from this member. We consider a community member as a reviewer, if our data contains at least one peer review activity from this member. Since members can assume both roles at the same time, we account for this overlap by assuming that contributors will not review

		LINUX	
Metric	ANDROID	**2005**	**2009**
Number of Community Members	739	2,300	4,901
Number of Reviewers	408	1,680	3,503
Number of Contributors	526	771	2,482
Number of Contributors with multiple submissions	203	475	1,792
Number of Contributors with multiple submissions who had at least one rejection	151	445	1,666
Number of Contributors who had a single submission that was rejected	37	208	405
Number of Contributors who returned to submit more contributions	38.5%	61.6%	72.19%

Table 7.4: Results of quantitative analysis of community activity in ANDROID and LINUX.

their own contributions (this is a strong assumption that might not hold true in reality, but it makes counting more feasible).

We present a summary of this categorization in Table 7.4. For LINUX 05, we measured the number of reviewers of community contributions and the number of contributors from our dataset and find that for each contributor, there are 2.17 reviewers, and that every contributor submits an average of 12 contributions (median: 2). Similarly, for LINUX 09 we find that for each contributor there are 1.41 reviewers, and that each contributor submits an average of 28.68 submissions (median: 4).

We statistically model the relation between the time needed for a decision to be made on a contribution (response), and the number of reviewers that worked together to arrive at that decision, while controlling for two factors: the size of the contribution (as larger contributions might take more effort to review), and the number of files a contribution adds or modifies as a proxy to the architectural complexity of the contribution (more complex contributions might involve more reviewers). A summary of our models for ANDROID and LINUX 09 is presented in Table 7.5.

Linux'05				
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	86575.4537	20917.9544	4.14	0.0000
contrib_size	-4.9482	9.9868	-0.50	0.6203
contrib_spread	-508.9277	377.9804	-1.35	0.1782
nr_reviewers	106395.0791	4426.8842	24.03	0.0000
Linux'09				
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-60193.5157	20389.2829	-2.95	0.0032
contrib_size	-24.3463	2.9830	-8.16	0.0000
contrib_spread	3259.0591	411.1269	7.93	0.0000
nr_reviewers	222711.3000	5347.0666	41.65	0.0000
Android				
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	353828.8584	144951.6108	2.44	0.0147
contrib_size	-15.4757	12.3464	-1.25	0.2101
contrib_spread	770.6971	140.3056	5.49	0.0000
nr_reviewers	230589.7729	36084.0235	6.39	0.0000

Table 7.5: Modeling Review Time in Linux'09 and Android by contribution size in number of lines of code (contrib_size), number of files added or modified by the contribution (file_spread), and the number of reviewers who were involved in making a final decision on whether to accept the contribution.

Overall, we observe that an increased number of reviewers is related to a significant increase in review time (keeping every other factor constant, adding an additional reviewer increases the total review time by 61 hours for `LINUX` 09 and 64 hours for `ANDROID`). We attribute this observation to the increased need for multiple reviewers to coordinate and reach a consensus on a final decision.

7.3.7 *Phases 4 and 5: Verification and Integration*

Verification

In `ANDROID`, verifiers, who are often senior (Google) engineers, who have been appointed by the leaders responsible for the individual `ANDROID` subsystems, merge the contribution into a local snapshot of the latest version of the code base, and manually test the contribution for correctness and functionality. If problems occur, the verifiers give feedback to the original contributor, who

can then resubmit an updated revision that addresses these problems. If verification succeeds, the contribution is accepted to ANDROID's development branch by the subsystem's lead reviewer or maintainer.

In LINUX, verification of the contribution takes place through developers and beta testers of the experimental branch. Feedback is provided to the original contributor through separate mailing lists that are dedicated to this testing purpose. If problems occur during this phase, contributions are put on hold until the issues are resolved.

Integration

In ANDROID, integration of contributions takes place as soon as they have been successfully verified. Project leads can initiate an automated integration process through the web interface of the code review system. If this automated merge is successful, the contribution is delivered to the community the next time they synchronize their local environments with the project directory.

Integration of contributions into the next release in LINUX is handled through a semi-automatic approach: contributions are manually integrated by developers, by moving them through development repositories until they finally reach the main development branch [Jiang2013]. However, to have a chance for being integrated in the upcoming version, a contribution has to be at least accepted during the merge window of the upcoming release. But even then, contributions can fail to make it, for example if the contribution is too risky to introduce at once or other features suddenly get higher priority. Linus Torvalds has the final say ("benevolent dictator") [Crowston2005] and can overrule all previous recommendations to reject contributions that do not fit with the strategic direction of LINUX.

We observe that in both systems some dedicated developers have the role of decision makers, who can ultimately deny the integration of a contribution into the master repository. ANDROID's integration strategy is much more immediate than the strategy eployed by LINUX, and delivers accepted contributions to the rest of the community without delay.

We believe our earlier observations on the importance of timely management hold for phases four and five as well. However, the issues are amplified: even though a contribution might have passed initial feedback and has been accepted through the review phase, long verification and integration processes may cause significant delays in follow-up communication with the original author of the contribution and may lead to already accepted contributions be abandoned at this late stage.

7.4 Threats to Validity

In the following we discuss the limitations of our study and the applicability of the results derived through our approach. For this purpose we discuss our work along four types of possible threats to validity [Yin2009]. In particular, these are: construct validity, internal validity, external validity and reliability.

7.4.1 Construct Validity

Threats to construct validity relate to evaluating the meaningfulness of the measurements used in our study and whether these measurements quantify what we want them to.

Our study contains a detailed case study of contribution management in `ANDROID` and `LINUX`. Within that case study, we quantify key characteristics of contribution management, along three dimensions, in particular, each dimension corresponds to a particular research question.

Within each dimension (activity of the community, size of contributions, timely management of contributions), we selected multiple measures that highlight the studied dimension from different angles. Our quantitative assessments are based on descriptive statistics about the

distributions of the collected data samples. When comparing findings across both projects, we support these comparisons through statistical hypothesis testing.

Time-span and trend data is studied through fitting of two regression models, first a linear regression model to observe the overall trends of the data in the studied time periods, and second a polynomial regression model to account for possible seasonal variation in our data, i.e., to counter bias introduced by software release cycles and software development processes in both projects.

In all cases where we performed multiple statistical significance tests, we used the Bonferroni correction [Rice1995] to avoid the spurious discovery of significant results due to multiple repeated tests.

With respect to the conceptual model of contribution management presented in the context of research question 1, threats to construct validity concern the extent to that our observations match reality. The conceptual model was systematically derived from multiple significant open-source software projects. All seven projects that were analyzed showed significant commonalities of managing contributions in a series of steps. Our conceptual model generalizes these steps into five distinct phases that contributions undergo in each project, before they become part of the software and are made available to the general public.

Within each of these phases however, there may exist processes and practices that are specific to a particular project. Our work presents two instances of the conceptual model which documents these processes and practices for two major open-source systems, `LINUX` and `ANDROID`. While we cannot claim completeness of the model (i.e., perfectly fitting the contribution management process of every open-source project in existence), we see our model as a first starting point for documenting and formalizing contribution management, in the same vein as architectural models have been derived and refined in the area of Software Architecture research.

7.4.2 *Internal Validity*

Threats to internal validity relate to the concern that there may be other plausible hypotheses explaining our findings.

We have carried out a detailed case study on how two related open source systems, ANDROID and LINUX do contribution management in practice. While ANDROID and the LINUX kernel share many similarities, for example drivers, device support, and core operating-system functionality that is added to the systems - we could argue that to some extent ANDROID contains LINUX. Even though both projects share similar roots, mindsets, tools, processes and domain, we believe that a comparison, such as carried out in our case study, is worthwhile and insightful. Our main goal is to study the management of contributions, rather than technical aspects or implementation details of the OS software.

We want to note that the scope of our study is not the technical aspects (or even content) of the individual contributions, but how contributions are managed in practice. With that respect, LINUX has switched to their current contribution management practices at about the same time as the development in ANDROID started. From documentation (e.g., the interview with Shawn Pearce referenced in our paper) we observed that ANDROID had looked closely at how LINUX manages contributions, and they consciously decided to pursue a different approach of contribution management that better suits their business.

Within our detailed case study on contribution management, we quantify and compare key characteristics of LINUX and ANDROID. Our quantitative findings and resulting hypotheses regarding the cause of these findings were followed up by detailed manual and qualitative study of the underlying data, to balance the threats for internal validity of our study.

7.4.3 *External Validity*

The assessment of threats to external validity evaluates to which extent generalization from the results of our study are possible.

In this study, we propose a conceptual model of contribution management. This model is an abstraction of the (commonalities in) contribution management processes of seven major open source projects. In the same way that architectural models create abstractions of actual instances of software implementations, our goal is to give researchers a common ground for conversation about, and further study of contribution management.

While the conceptual contribution management model was derived from only a tiny fraction of open source projects in existence, we still argue that generalizability of the model is high. The conceptual model was derived through a systematic approach, known as "Grounded Theory" [Glaser1967; Strauss1990], of publicly accessible records of processes and practices of seven projects contemporary, prominent and important open-source projects.

While we can claim neither completeness, nor absolute correctness of the derived model, our abstraction serves as a starting point for an abstraction of contribution management on a scientific basis, and we hope that future research will lead to incremental refinements of this abstraction, similar to conceptual models of software architecture.

Our detailed case study on `LINUX` and `ANDROID`, illustrates real instantiations of contribution management in practice, and details key characteristics along three dimensions. Due to the nature of this study, our observations are bound to the two studied systems, and unlikely to generalize to the broad spectrum of open source projects in existence. However, we report on a variety of practices that are found in many open source projects, such as "cherry-picking", i.e., selecting only small parts of a contribution for inclusion into the project, and outline problems, for example, the re-implementation of functionality by multiple contributors at the same time. As such our case study stands as a report of examples in "best-practices" and potential "pitfalls"

in two large and mature open source systems, which are likely to be applicable across a broader range of domains and other open source projects.

7.4.4 *Reliability*

The assessment of threats to the reliability of our study evaluates the degree to which someone analyzing the data presented in this work would reach the same results or conclusions.

We believe that the reliability of our study is very high. Our conceptual model of contribution management is derived from publicly available documentation and data of seven large open-source software systems. Furthermore, our case studies on how the `LINUX` kernel and `ANDROID` OS projects carry out contribution management in practice rely on data mined from publicly available data repositories (email archives in the case of `LINUX`, and `GERRIT` data in the case of `ANDROID`). The methods used to collect that data are described in detail in Section 2, and have also been used in previous research, e.g., the work by Jiang et al. [Jiang2013].

7.5 *Related Work*

The work presented in this chapter is related to a variety of previous studies on open source development processes, which we discuss in the following.

7.5.1 *Community-Driven Evolution through Code Contributions*

In their work "The Cathedral and the Bazaar", Raymond et al. [Raymond2001] discussed core ideas behind the success of the open source movement. Raymond's main observations are

formulated as Linus' law, i.e., the more people reviewing a piece of code, the more bugs can be found, and on the motivation of developers to add the features they are interested in.

While our work does not attempt to prove or disprove Linus' law, we have studied two systems that strongly support and encourage user-driven evolution, but are substantially different in the way they treat user contributions and merge them into their own code-bases. For instance, in ANDROID, contributions do need to align with the strategic goals of the leaders for the module these contributions target, a community interest in a feature alone is not sufficient for inclusion.

For example, ANDROID community contribution #11758 was rejected despite large community interest (in ANDROID, users can "star" a proposed contribution if they are enthusiastic about having that feature added) for the contributed feature, in particular one reviewer of the contribution[10] notes:

> *"It might have 40 stars but you have to weigh in the cost of adding a rather obscure/technical UI preference for *everybody* for the benefit of a few."*

7.5.2 *Community-Driven Development as a Business Model*

Both Hecker [Hecker1999], and Krishnamurty [Krishnamurthy2005] provided a comprehensive overview and analysis of modern open source business models. While Hecker outlined potential pitfalls that businesses have to be aware of when moving towards these models, Krishnamurty described key factors for the success of open source business models.

In particular, Hecker et al.'s work [Hecker1999] puts a large emphasis on the importance of the community in the open source business model. Our work extends on previous knowledge through our qualitative study on how two major and successful open source projects handle business concerns like intellectual property management, peer review, and the potential risk of meaningless contributions.

[10] https://android-review.googlesource.com/#/c/11758/

7.5.3 *Community-Contribution Management*

Mockus et al. [Mockus2002] investigated email archives and source code repositories of the APACHE and Mozilla projects, to quantify the development processes and compare them to commercial systems. They found that APACHE has a democratic contribution management process, consisting of a core group of developers with voting power and CVS access, and a contributor community of 400 developers. In addition, Mockus et al. identified Mozilla as having a "hybrid" process, since it was spawned from the commercial Netscape project.

Furthermore, Mockus et al. conjectured that "it would be worth experimenting, in a commercial environment, with OSS-style open work assignments". Our work extends on this notion through the systematically derived coneptual model of contribution management, as well as our qualitative and quantitative investigation of two concrete instances of that conceptual model.

Rigby et al. [Rigby2008] investigated peer review practices in the APACHE project, and compared these to practices observed in a commercial system. They found that small, independent, complete contributions are most successful, and that the group of actual reviewers in a system is much smaller than potentially is achievable. The work by Rigby et al. focused on the time component of reviewing large and small contributions, while our work investigates acceptance bias.

Capiluppi et al. [Capiluppi2003] studied the demography of open source systems, and found that few projects are capable of attracting a sizeable community of developers. In particular, Capiluppi et al. found that 57% of the projects consist of 1 or two developers, with only 15% having 10 or more developers, leading to slow development progress. In addition, Capiluppi et al. remarked that larger projects typically have 1 co-ordinator for every 4 developers. In contrast to their work, we find that for both LINUX and ANDROID there is a significantly higher ration of co-ordinators to developers.

Weissgerber et al. [Weissgerber2008] studied patch contributions in two open source systems, and found that 40% of the contributions are accepted. Contrary to our findings, they find

that smaller patches have a higher probability of being accepted.

Crowston et al. [Crowston2005] examined 120 open source project teams, and found that their organization ranges from dictatorship-like projects to highly decentralized structures. Our work presents two instances of organizatorial structure: `LINUX` as a hierarchy of individuals with increasing power over the ultimate decisions connected with community contributions, and `ANDROID`, as a decentralized, vote-based structure.

Within the same vein of the work presented in this study, Sethanandha et al. [Sethanandha2010] proposed a framework for the management of open source patch contributions which they derived from 10 major open source projects. While the work of Sethanandha et al. focussed more on the handling of patches and their application against the code base, our work paints a more general picture of contribution management, which spans from inception to integration of a contribution. In addition, we carry out a detailed case study on two open source ecosystems to investigate how the process is implemented in practice.

7.6 *Conclusions*

Contribution management is a real-world problem that has received very little attention from the research community so far. Even though many potential issues with accepting contributions from external developers (i.e., developers who are not part of an in-house development team) into a software project have been outlined in literature, little is known on how these issues are tackled in practice. While deriving a conceptual model of contribution management from seven major open source projects, we found that even though projects seem to follow a common set of steps – from the original inception of a contribution to the final integration into the project codebase – the different contribution management practices are manifold and diverse, often tailored towards the specific needs of a project.

Even though both studied systems (LINUX and ANDROID) employ different strategies and techniques for managing contributions, our results show that both approaches are valuable examples for practitioners. However, each approach has specific advantages and disadvantages that need to be carefully evaluated by practitioners when adopting either contribution management process in practice. While a more open contribution management process, such as the one employed by LINUX, reduces the overall management effort by giving control over the process to a self-organized community. Disadvantages of such self-organized and community driven contribution management include weaker intellectual property management, and the risk of missed contributions. A more controlled contribution management system, such as the one employed by ANDROID overcomes these advantages, at the cost of an increased management effort that, depending on the size of the community, might burn out the reviewers, since they are forced to review changes and follow up on eventual revisions.

The findings of our quantitative assessment of the contribution management processes of both projects, makes a case for the importance of timely feedback and decisions to avoid losing valuable contributions. In contrast to LINUX, where even a more than three times increase of feedback delay does not seem to be a cause for alarm, we found that ANDROID makes active efforts to decrease feedback times, with the goal to foster increased interactivity with the community.

For future work, we aim to extend the conceptual model by studying additional open source projects, and their contribution management practices. In particular, we plan to extend our contribution management model with the pull-request process popularized by GIT providers such as GitHub and BitBucket. Furthermore, we plan to carry out an in-depth investigation of the key factors that influence attraction and retention of community members. Third, we plan to study the impact of the different decision practices in LINUX and ANDROID on project planning and feature integration.

Part IV

Conclusions and Future Work

8

Conclusions and Future Work

In software development, the knowledge of developers, architects and end users is spread out across dozens of development artifacts. Historically, structured development artifacts such as source code have been the primary focus of software engineering research, and have formed the main foundation of how we understand software quality today.

In this thesis, we argue that understanding software quality requires more than reverse-engineering source code. Developer communication captured in bug reports, execution logs, mailing lists, code review reports, change log messages and requirements documents contains a significant amount of implicit developer knowledge on the software and provide valuable information about the social aspect of software development. These records of developer communication mainly consist of unstructured data: a mix of natural language text and technical information about the software. Mining unstructured data is challenging, since traditional parsing and extraction techniques typically cannot handle free-form data well or identify structured components in unstructured data.

However, using off-the-shelf techniques to process this data naïvely yields many risks for

the validity of the resulting information. Many of the text cleaning techniques used in text-mining and information retrieval cannot be readily applied to communication data. The pitfalls when mining communication data, together with the suggestions to avoid these pitfalls outlined in this thesis enables practical mining of communication data. Our lightweight approach to finding technical information in communication data, goes one step further, and makes technical information readily available for researchers and practitioners.

Based on the communication data that we mined with the tools and techniques developed in the first part of this thesis, we were able to establish a set of socio-technical metrics that we demonstrate to be equally valuable for studying software quality, as traditional product and process metrics. In addition, our findings demonstrate that our novel metrics can be used to complement traditional models, as we are thus able to obtain more powerful defect prediction models. Our findings confirm the value of socio-technical information in the software engineering domain, and make a strong case in favor of the importance of the social side of software engineering. Based on our results, we believe that socio-Technical information provides a more complete picture of the factors influencing the quality of a software product. In particular, the strong empirical evidence of the value that socio-technical information adds to models validates the hypothesis that software engineering is a highly social process.

Lastly, the work presented in this thesis makes a strong case for the importance of effective communication between developers and the users surrounding a software, in the context of open-source business models, for building a healthy product halo. Our detailed reports on two instances of collaboration between developers and volunteer developers from the user community show that the overarching theme of communication as a key factor of success holds true for software development. We hope that this thesis makes a compelling case for the importance of future improvements in the way how developers communicate not only with each other but also with the volunteer developers who are part of the community surrounding the software, as a major factor for the successful evolution of the software.

8.1 Thesis Contributions and Findings

This thesis makes a variety of contributions to the research field. The technical contributions of this thesis center around the development and evaluation of tools and techniques for mining communication data, which exists mainly as unstructured data and is unsuitable for processing with off-the-shelf data mining approaches. The conceptual contributions of this thesis center around the exploration of developer communication as a key factor for the quality of a software. The empirical contributions of this thesis are the application of the proposed tools and techniques for mining communication data from several long-lived open source projects, as well as a documentation of the effects of developer communication on software quality.

The main contributions of this thesis are as follows:

- A systematic literature review on the state-of-the-art of research on relationships between socio-technical information about the software development process and the quality of the software.
- The concept of communication data as unstructured data, which cannot be readily processed by traditional data mining, information retrieval and natural language processing approaches. Communication data exists as unstructured data that intertwines natural language text, project specific language, automated text, and technical information.
- The documentation of the details of common problems as well as possible technical solutions for mining unstructured data. We demonstrate that improper handling of this special kind of data can lead to substantial bias in data, experiments and results.
- The implementation of an email-mining tool named MailboxMiner[1], which is publicly available and has found extensive use in the research area.

[1] https://github.com/nicbet/MailboxMiner

- A light-weight approach for separating technical information from natural language text in unstructured data. Additionally, as part of this work, a manually developed benchmark suite to evaluate and compare the performance of future approaches against.

- An approach to link communication data to those parts of the source code that are being discussed. Different conceptual classes of links between communication data and the software can be established. We show that the approach presented in this thesis produces a novel class of traceability links that are more suitable for socio-technical analyses then traditional approaches.

- A novel set of socio-technical metrics surrounding the social interactions between developers. We show that three dimensions of socio-technical relationships exist that we can measure from developer communication. We demonstrate that these metrics can explain software defects as well as traditional source-code based metrics. In addition, we show that a combination of these socio-technical metrics and traditional product and process metrics in defect models, yields higher explanatory power than taken separately.

- A conceptual model of contribution management, as well as an investigation of the role of developer communication in the context of contribution management, which can be used by practitioners who aim at establishing effective contribution management when moving towards an open source business model. Through a case study on two large open source software systems we document that ineffective communication systems can have negative effects on community-contributed source code and thus the successful evolution of the software.

8.2 Suggestions for Extending this Thesis

We believe that our thesis makes a positive contribution towards providing empirical evidence of the strong relationships between developer communication and the quality, as well as the evolution of a software product. However, we believe that our work also opens a number of research opportunities. In the following, we highlight potential future work to extend our results.

8.2.1 Exploration of Additional Communication Repositories

In Chapter 3, we demonstrated tools and techniques for mining communication data form email repositories. While e-mail remains the most popular means of asynchronous communication between developers, engineers use a variety of channels, such as formal weekly meetings, informal meetings like face-to-face chat in hallways and kitchens, electronic chat applications, phone, or social media [Wu2003]. As such, email repositories capture only a fraction of the communication surrounding the development process, which may contain only a partial view on the collaborative activities of developers. We believe that further exploration of tools and techniques for capturing communication through these additional channels prove valuable for obtaining a more complete and broader view on the central role of communication in software development to facilitate collaborative activities.

8.2.2 Understanding the significance of technical information within the communication between developers

In Chapter 4, we presented a lightweight approach for separating technical information from natural language text in communication data. We used that technical information in Chapter 5 to link communication data to the parts of the software that is being talked about. However, we

believe that we need to look at a broader picture and investigate the reasons behind communicating that particular piece of information. A study by Bacchelli et al. [Bacchelli2010a] presents a first step in this direction, investigating whether software modules that are being frequently mentioned in developer communication are more likely to contain errors.

8.2.3 *Capturing Additional Aspects of Communication Through Social Metrics*

In Chapter 6, we presented a variety of socio-technical metrics surrounding the communication between developers. However, this selection of metrics is far from complete and while we demonstrate that the underlying statistical models explain software defects as well as source code metrics, the factors we captured and used as predictor variables in those models may be incomplete. To counter potential bias due to not capturing important causal factors, and to gain a better understanding of what aspects of communication influence software quality, we believe that research would greatly benefit from further investigation of qualitative aspects of communication and how we can measure these aspects from communication data.

8.3 *Opportunities for Future Research*

The aforementioned research opportunities are specific to the context of the research work presented in this thesis. In addition, our literature review presented in Chapter 2 suggests a number of future research avenues that are specific to the broader research area, and that we summarize in the following.

8.3.1 *Understanding the Semantics of Social Network Analysis Metrics*

We observed that SNA metrics are interpreted specific to the context of each study. However, we are not aware of any attempt to collect, summarize and put possible interpretations of the relationships between SNA metrics and software quality aspects into a common framework. We believe that further investigation of SNA metrics obtained from socio-technical networks and an attempt at generalization may provide deeper insights into the coordination of development teams, the setup of software development processes and the architectural organization of the source code.

8.3.2 *Extending the Concept of Social-Technical Congruence Beyond File-Developer Networks*

We observed that Socio-Technical Congruence is largely unexplored beyond validation of Conway's law. We believe that the concept of socio-technical congruence can be applied to a broader range of socio-technical networks, beyond the file-developer networks described in Conway's law, and might provide a general measure of how well software development concerns overlap.

8.3.3 *Investigating the Impact of Co-Location on Software Quality in Open Source Projects*

We have observed conflicting empirical evidence on the effect of co-location on software quality in the industrial domain. We believe further investigations of the impact of co-location, and in particular insights in how open-source projects, which by their very nature are developed by widely distributed teams, successfully cope with co-location concerns, might provide valuable knowledge about successful processes and practices that could be adapted in industrial settings.

8.3.4 *Investigating a Broader Range of Quality Metrics*

We observed that a majority of research focuses on readily available and measurable quality metrics such as the number of defects delivered to the customer, or development effort spent removing defects. However, little empirical evidence is provided by existing literature on less readily available software quality concerns, such as project health, project profit, as well as implicit quality aspects which might strongly influence future development, such as software design quality, deviations from coding standards, or documentation of the source code. We believe that further investigations on the relationships between socio-technical concerns and these largely unexplored quality aspects might yield valuable insights for practitioners and researchers alike.

8.4 *Closing Remarks*

Software development is a complex symphony of a broad range of development tasks, ranging from design over documentation to the actual encoding of logic in the software's source code. Effective software development thus requires the coordination of these activities among developers to avoid breaking the source code and introducing errors into the software.

At the heart of coordination among a group of developers stands communication between individuals, together with all the social aspects of human interaction such communication brings with it. We believe that the realization of software development as a highly social process, in which the social aspects of interactions between developers play a key factor in the quality of a software is likely to take on a central role in future software engineering research.

Our work contributes to the field of software engineering by demonstrating that software repositories contain a wealth of developer communication captured in a variety of artifacts produced throughout the software development process, which can be used by practitioners and

researchers to obtain a second view on software, which is orthogonal to the traditional views that rely solely on technical facts about the source code and the process employed to create that code.

We hope that this thesis will encourage research to explore integrating social aspects of software development in addition to traditional technical views in their analyses. Our goal is to entice practitioners to consider the impact social interactions between developers and development teams can have on the quality of their software product, with the aim to development environments that stimulate effective developer communication.

List of References

[Alatis1993] James E. Alatis. *Language, communication and social meaning*. Georgetown University Press, 1993, p. 507 (cited on page 167).

[Amrit2005] Chintan Amrit. "Application of social network theory to software development: The problem of task allocation". In: *Proceedings of the 2nd International Workshop on Computer Supported Activity Coordination, CSAC 2005, in Conjunction with ICEIS 2005, May 24, 2005 - May 25, 2005*. Miami, FL, United states: INSTICC Press, 2005, pp. 3–13 (cited on pages 44, 45, 54, 55, 58–61, 63, 66, 69, 70, 72–75).

[Amrit2010] Chintan Amrit and Jos van Hillegersberg. "Exploring the Impact of Socio-Technical Core-Periphery Structures in Open Source Software Development". In: *CoRR* abs/1006.1244 (2010) (cited on pages 35, 48, 49, 55, 58, 60, 63, 67, 69, 72, 74, 75).

[Antoniol2000] G. Antoniol et al. "Tracing Object-Oriented Code into Functional Requirements". In: *Proceedings of the 8th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2000, p. 79 (cited on pages 126, 129, 146).

[Antoniol2002] Giuliano Antoniol et al. "Recovering Traceability Links between Code and Documentation". In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983 (cited on pages 117, 126, 130, 131, 144).

[Antoniol2008] Giuliano Antoniol et al. "Is it a bug or an enhancement?: A text-based approach to classify change requests". In: *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research*. Ontario, Canada: ACM, 2008, pp. 304–318 (cited on page 173).

[Anvik2006] John Anvik, Lyndon Hiew, and Gail C. Murphy. “Who should fix this bug?” In: *Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China: ACM, 2006, pp. 361–370 (cited on pages 117, 168).

[Aranda2009] Jorge Aranda and Gina Venolia. “The secret life of bugs: Going past the errors and omissions in software repositories”. In: *Proceedings of the 31st International Conference on Software Engineering*. 2009, pp. 298–308 (cited on pages 15, 71).

[Asuncion2010] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. “Software Traceability with Topic Modeling”. In: *Proceedings of the 32nd International Conference on Software Engineering*. IEEE Computer Society, 2010, pp. 95–104 (cited on pages 131, 144).

[Asundi2007] Jai Asundi and Rajiv Jayant. “Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study”. In: *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2007, p. 166c (cited on page 220).

[Bacchelli2009] Alberto Bacchelli et al. “Benchmarking Lightweight Techniques to Link E-Mails and Source Code”. In: *Proceedings of the 16th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 205–214 (cited on pages 126, 131–133).

[Bacchelli2010a] Alberto Bacchelli, Marco D’Ambros, and Michele Lanza. “Are popular classes more defect prone?” In: *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, March 20, 2010 - March 28, 2010*. Vol. 6013 LNCS. Paphos, Cyprus: Springer Verlag, 2010, pp. 59–73 (cited on pages 15, 24, 37, 38, 53, 55, 58, 60, 63, 68, 69, 72, 74, 75, 117, 126, 206, 264).

[Bacchelli2010b] Alberto Bacchelli, Marco D’Ambros, and Michele Lanza. “Extracting Source Code from E-Mails”. In: *Proceedings of the 18th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2010, pp. 24–33 (cited on pages 114, 117, 122).

[Bacchelli2010c] Alberto Bacchelli, Michele Lanza, and Romain Robbes. “Linking E-Mails and Source Code Artifacts”. In: *Proceedings of the 32nd International Conference on Software Engineering*. IEEE Computer Society, 2010, pp. 375–384 (cited on pages 131, 136).

[Baldwin1999] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. Cambridge, MA, USA: MIT Press, 1999 (cited on page 2).

[Barbagallo2008] Donato Barbagallo, Chiara Francalanci, and Francesco Merlo. "The impact of social networking on software design quality and development effort in open source projects L'impact des reseaux sociaux sur la qualite de la conception logicielle et l'effort de developpement dans les projets de logiciels libres". In: *29th International Conference on Information Systems, ICIS 2008, December 14, 2008 - December 17, 2008*. Paris, France: Association for Information Systems, 2008, Microsoft, IBM, Orange, GS1 France, Ernst and Young, et al (cited on pages 35, 44, 45, 55, 58–60, 63, 65, 66, 69, 72, 74, 75).

[Barbagallo2009] Donato Barbagallo and Chiara Francalanci. "The relationship among development skills, design quality, and centrality in open source projects". In: *ECIS*. Verona, Italy: Association for Information Systems, 2009 (cited on pages 35, 46, 48, 53, 55, 58, 60, 63, 65, 69, 71, 72, 74, 75).

[Basili1996] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators". In: *IEEE Transactions on Software Engineering* 22 (10 1996), pp. 751–761 (cited on pages 13, 205).

[Begel2010] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. "Codebook: discovering and exploiting relationships in software repositories". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa: ACM, 2010, pp. 125–134 (cited on page 2).

[Bettenburg2008a] Nicolas Bettenburg. "Duplicate Bug Reports Considered Harmful?" MA thesis. Faculty of Natural Sciences and Technology, Saarbruecken, Germany: Saarland University, July 2008 (cited on page 135).

[Bettenburg2008b] Nicolas Bettenburg et al. "Extracting Structural Information from Bug Reports". In: *Proceedings of the 5th International Workshop on Mining Software Repositories*. Leipzig, Germany, May 2008 (cited on pages 114, 117, 135, 138, 161, 207).

[Bettenburg2008c] Nicolas Bettenburg et al. "What makes a good bug report?" In: *Proceedings of the 2008 ACM SIGSOFT Symposium on Foundations of Software Engineering*. Atlanta, Georgia: ACM, 2008, pp. 308–318 (cited on pages 161, 207).

[Bettenburg2010a] N. Bettenburg and A. E. Hassan. “Studying the Impact of Social Structures on Software Quality”. In: *Proceedings of the 18th IEEE International Conference on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 124–33 (cited on pages 24, 35, 37, 38, 53, 55, 58, 60, 62–65, 69, 70, 72, 74–77, 86, 126).

[Bettenburg2010b] Nicolas Bettenburg and Bram Adams. “Workshop on Mining Unstructured Data (MUD) because "Mining Unstructured Data is Like Fishing in Muddy Waters"!” In: *Proceedings of the 2010 Working Conference on Reverse Engineering* (2010), pp. 277–278 (cited on pages 15, 83, 114).

[Bettenburg2013a] Nicolas Bettenburg et al. “Management of community contributions”. In: *Empirical Software Engineering* (2013), pp. 1–38 (cited on page 86).

[Bettenburg2013b] Nicolas Bettenburg et al. “Management of community contributions”. In: *Empirical Software Engineering* (2013), pp. 1–38 (cited on page 216).

[Bicer2011] Serdar Bicer, Ayse Basar Bener, and Bora Caglayan. “Defect prediction using social network analysis on issue repositories”. In: *Proceedings of the 2011 International Conference on Software and Systems Process, ICSSP 2011*. Waikiki, Honolulu, HI, United states: IEEE Computer Society, 2011, pp. 63–71 (cited on pages 35, 49, 55, 58, 60, 61, 63, 64, 69, 72, 74, 75).

[Bird2006a] Christian Bird et al. “Mining email social networks”. In: *Proceedings of the 3rd International Workshop on Mining Software Repositories, MSR ’06, Co-located with the 28th International Conference on Software Engineering, ICSE 2006*. Shanghai, China: IEEE Computer Society, 2006, pp. 137–143 (cited on pages 35, 45, 46, 55, 58, 60, 61, 63, 67, 69, 72, 74, 75, 107, 109).

[Bird2006b] Christian Bird et al. “Mining email social networks in Postgres”. In: *Proceedings of the 3rd International Workshop on Mining Software Repositories, MSR ’06, Co-located with the 28th International Conference on Software Engineering, ICSE 2006*. Shanghai, China: ACM, 2006, pp. 185–186 (cited on pages 17, 109).

[Bird2007] Christian Bird, Alex Gourley, and Prem Devanbu. “Detecting Patch Submission and Acceptance in OSS Projects”. In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 26 (cited on page 226).

[Bird2008] Christian Bird et al. “Latent Social Structure in Open Source Projects”. In: *Proceedings of the 2008 ACM SIGSOFT Symposium on the Foundations*

of Software Engineering. Atlanta, Georgia, USA, 2008, pp. 24–35 (cited on pages 3, 14).

[Bird2009a] Christian Bird et al. “Does distributed development affect software quality? An empirical case study of windows vista”. In: *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, May 16, 2009 - May 24, 2009*. Vancouver, BC, Canada: IEEE Computer Society, 2009, pp. 518–528 (cited on pages 2, 35, 41–43, 55, 57, 58, 60, 62, 63, 69, 70, 72, 74, 75).

[Bird2009b] Christian Bird et al. “Fair and balanced?: bias in bug-fix datasets”. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Amsterdam, The Netherlands: ACM, 2009, pp. 121–130 (cited on pages 131, 132, 136, 139, 157, 161, 174, 202).

[Bird2009c] Christian Bird et al. “Putting it all together: Using socio-technical networks to predict failures”. In: *Proceedings of the 20th International Symposium on Software Reliability Engineering, ISSRE 2009*. Mysuru, Karnataka, India: IEEE Computer Society, 2009, pp. 109–119 (cited on pages 35, 47, 48, 53, 55, 58, 60, 63, 69, 72, 74, 75).

[Bird2011] Christian Bird. “Sociotechnical coordination and collaboration in open source software”. In: *Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM 2011, September 25, 2011 - September 30, 2011*. Williamsburg, VA, United states: IEEE Computer Society, 2011, pp. 568–573 (cited on page 43).

[Bird2012] C. Bird and N. Nagappan. “Who? Where? What? Examining distributed development in two large open source projects”. In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR 2012), 2-3 June 2012*. Piscataway, NJ, USA: IEEE, 2012, pp. 237–46 (cited on pages 35, 43, 55, 58, 60, 63, 69, 71, 72, 74, 75).

[Bland1996] Martin J Bland and Douglas G Altman. “Transformations, means and confidence intervals.” In: *British Medical Journal* 312.7038 (1996), p. 1079 (cited on page 175).

[Boulos0703] M.N.K. Boulos and S. Wheeler. “The emerging Web 2.0 social software: an enabling suite of sociable technologies in health and health care education”. English. In: *Health Inf. Libr. J. (UK)* 24.1 (2007/03/), pp. 2–23 (cited on page 26).

[Buckley1984] Fletcher J. Buckley and Robert M. Poston. “Software Quality Assurance”. In: *IEEE Transactions on Software Engineering* 10 (1 1984), pp. 36–41 (cited on page 13).

[Budgen2006] David Budgen and Pearl Brereton. “Performing systematic literature reviews in software engineering”. In: *Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, pp. 1051–1052 (cited on page 20).

[Canfora2011] Gerardo Canfora et al. “Social interactions around cross-system bug fixings: The case of FreeBSD and OpenBSD”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR 2011, Co-located with ICSE 2011*. Waikiki, Honolulu, HI, United states: IEEE Computer Society, 2011, pp. 143–152 (cited on pages 35, 50, 55, 58, 60, 63, 69, 72, 74, 75).

[Capiluppi2003] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. “Characteristics of Open Source Projects”. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 317 (cited on page 254).

[Carvalho2004] Vitor Rocha de Carvalho and William W. Cohen. “Learning to Extract Signature and Reply Lines from Email”. In: *CEAS*. 2004 (cited on pages 104, 109).

[Cataldo2006] Marcelo Cataldo et al. “Identification of coordination requirements: Implications for the Design of collaboration and awareness tools”. In: *Proceedings of the 20th Anniversary ACM Conference on Computer Supported Cooperative Work, CSCW 2006, November 4, 2006 - November 8, 2006*. Banff, AB, Canada: Association for Computing Machinery, 2006, pp. 353–362 (cited on pages 12, 14, 35, 51, 55, 58–60, 63, 68–72, 74, 75).

[Cataldo2008a] Marcelo Cataldo and James D. Herbsleb. “Communication patterns in geographically distributed software development and engineers’ contributions to the development effort”. In: *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE ’08, Co-located with the 13th International Conference on Software Engineering, ICSE 2008, May 10, 2008 - May 18, 2008*. Leipzig, Germany: IEEE Computer Society, 2008, pp. 25–28 (cited on pages 24, 35, 40, 42, 55, 57, 58, 60, 61, 63, 67, 69, 72, 74, 75).

[Cataldo2008b] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. “Socio-technical congruence: a framework for assessing the impact of technical

and work dependencies on software development productivity". In: *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Kaiserslautern, Germany: ACM, 2008, pp. 2–11 (cited on pages 2, 3, 14, 34, 35, 51, 55, 58, 60, 63, 65, 67, 69, 70, 72, 74, 75).

[Cataldo2009a] Marcelo Cataldo and Sangeeth Nambiar. "On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality". In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Amsterdam, The Netherlands: ACM, 2009, pp. 101–110 (cited on pages 13, 35, 42, 43, 53–55, 58, 60, 63, 69, 72, 74, 75).

[Cataldo2009b] Marcelo Cataldo et al. "Software Dependencies, Work Dependencies, and Their Impact on Failures". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 864–878 (cited on pages 3, 13, 14, 157, 158, 171, 172).

[Cataldo2011] Marcelo Cataldo and James D. Herbsleb. "Factors leading to integration failures in global feature-oriented development: an empirical analysis". In: *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 161–170 (cited on pages 35, 42, 43, 55, 58, 60, 63, 64, 69, 72, 74, 75).

[Charmaz2006] K. Charmaz. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. Constructing grounded theory. SAGE Publications, 2006 (cited on page 219).

[Cheluvaraju] Bharath Cheluvaraju, Kartikay Nagal, and Anjaneyulu Pasala. "Mining software revision history using advanced social network analysis". In: *Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC 2012, December 4, 2012 - December 7, 2012*. Vol. 1. IEEE Computer Society, pp. 717–720 (cited on page 19).

[Chidamber1994] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493 (cited on pages 56, 153).

[Cohen2003] J. Cohen. *Applied Multiple Regression/correlation Analysis for the Behavioral Sciences*. Applied Multiple Regression/correlation Analysis for the Behavioral Sciences v. 1. Routledge, 2003 (cited on page 171).

[Conway1968] M.E. Conway. "How do committees invent". In: *Datamation* 14.4 (1968), pp. 28–31 (cited on pages 2, 14).

[Crocker1982] David H. Crocker. *Standard for the Format of ARPA Internet Text Messages*. RFC 822 (Standard). 1982 (cited on page 104).

[Crowston2005] Kevin Crowston and James Howison. "The social structure of Free and Open Source software development". In: *First Monday* 10.2 (2005) (cited on pages 247, 255).

[Cubranic2003] Davor Ć Čubranić and Gail C. Murphy. "Hipikat: recommending pertinent software development artifacts". In: *Proceedings of the 25th International Conference on Software Engineering*. Portland, Oregon: IEEE Computer Society, 2003, pp. 408–418 (cited on pages 86, 174, 208).

[Cubranic2005] Davor Ć Čubranić et al. "Hipikat: A Project Memory for Software Development". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465 (cited on pages 130, 131).

[Dawson2003] Ray Dawson and Bill O'neill. "Simple Metrics for Improving Software Process Performance and Capability: A Case Study". In: *Software Quality Journal* 11 (3 2003), pp. 243–258 (cited on page 13).

[DEste2004] Claire D'Este. "Sharing Meaning with Machines". In: *Proceedings of the 4th International Workshop on Epigenetic Robotics*. Lund University Cognitive Studies, 2004, pp. 111–114 (cited on page 167).

[DiPenta] Massimiliano DiPenta et al. "The effect of communication overhead on software maintenance project staffing: A search-based approach". In: *Proceedings of the 23rd International Conference on Software Maintenance, ICSM*. IEEE Computer Society, pp. 315–324 (cited on page 19).

[Ducheneaut2005] Nicolas Ducheneaut. "Socialization in an Open Source Software Community: A Socio-Technical Analysis". In: *Comput. Supported Coop. Work* 14 (4 Aug. 2005), pp. 323–368 (cited on page 14).

[Duvall2007] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007 (cited on page 225).

[Edwards1963] A. W. F. Edwards. "The Measure of Association in a 2 by 2 Table". In: *Journal of the Royal Statistical Society. Series A (General)* 126.1 (1963), pp. 109–114 (cited on page 172).

[Espinosa2002] J. Alberto Espinosa. "Shared Mental Models and Coordination in Large-scale, Distributed Software Development". AAI3065743. PhD thesis. Pittsburgh, PA, USA, 2002 (cited on pages 12, 45).

[Espinosa2007] J. Alberto Espinosa et al. “Familiarity, Complexity, and Team Performance in Geographically Distributed Software Development”. In: *Organization Science* 18.4 (2007), pp. 613–630 (cited on pages 35, 40, 53, 55, 57, 58, 60, 63, 69, 72, 74, 75).

[Eyolfson2011] Jon Eyolfson, Lin Tan, and Patrick Lam. “Do time of day and developer experience affect commit bugginess”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR 2011, Co-located with ICSE 2011*. Waikiki, Honolulu, HI, United states: IEEE Computer Society, 2011, pp. 153–162 (cited on pages 16, 35, 38, 54, 55, 58, 60, 62, 63, 69, 70, 72, 74, 75).

[Fay2010] Michael P. Fay and Michael A. Proschan. “Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules”. In: *Statistics Surveys* 4 (2010), pp. 1–39 (cited on page 200).

[Fenton1991] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. London, UK, UK: Chapman and Hall, Ltd., 1991 (cited on page 13).

[Fischer2003a] Michael Fischer, Martin Pinzger, and Harald Gall. “Analyzing and Relating Bug Report Data for Feature Tracking”. In: *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE ’03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 90 (cited on pages 117, 126, 131, 174).

[Fischer2003b] Michael Fischer, Martin Pinzger, and Harald Gall. “Populating a Release History Database from Version Control and Bug Tracking Systems”. In: *Proceedings of the International Conference on Software Maintenance (ICSM ’03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 23 (cited on page 131).

[Frakes1987] W B Frakes and B A Nejmeh. “Software reuse through information retrieval”. In: *SIGIR Forum* 21.1-2 (1987), pp. 30–36 (cited on page 129).

[Freed1996a] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*. RFC 2049 (Draft Standard). Nov. 1996 (cited on pages 88, 100).

[Freed1996b] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045 (Draft Standard). Nov. 1996 (cited on pages 88, 100).

[Freed1996c] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046 (Draft Standard). Nov. 1996 (cited on pages 88, 100).

[Freed1996d] N. Freed, J. Klensin, and J. Postel. *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*. RFC 2048 (Best Current Practice). Nov. 1996 (cited on pages 88, 100).

[Friendly2002] Michael Friendly. "Corrgrams: Exploratory Displays for Correlation Matrices". In: *The American Statistician* 56.1 (Nov. 2002), pp. 316–324 (cited on page 176).

[Galhardas2000] Helena Galhardas et al. "Declaratively Cleaning your Data with AJAX". In: *BDA*. Ed. by Anne Doucet. 2000 (cited on page 98).

[German2009] Daniel M. German and Ahmed E. Hassan. "License integration patterns: Addressing license mismatches in component-based development". In: *Proceedings of the 31st International Conference on Software Engineering*. 2009, pp. 188–198 (cited on pages 222, 223).

[German2013a] Daniel M German, Bram Adams, and Ahmed E Hassan. "The evolution of the R software ecosystem". In: *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE. 2013, pp. 243–252 (cited on page 6).

[German2013b] D.M. German, B. Adams, and A.E. Hassan. "The Evolution of the R Software Ecosystem". In: *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*. Mar. 2013, pp. 243–252 (cited on pages 85, 110).

[Giger] E. Giger, M. Pinzger, and H. C. Gall. "Can we predict types of code changes? An empirical analysis". In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR 2012)*. IEEE, pp. 217–26 (cited on page 20).

[Glaser1967] Barney G Glaser and Anselm L Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967, p. 271 (cited on pages 219, 251).

[GNOME2009] GNOME. *GNOME Mailing List Archives*. http://mail.gnome.org/archives/. Last visited March 2009. 2009 (cited on page 92).

[Google2009] Google. *GMail*. http://mail.google.com. Last visited March 2009. 2009 (cited on page 103).

[Greenhalgh2005] Trisha Greenhalgh and Richard Peacock. “Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources”. In: *BMJ* 331.7524 (Nov. 2005), pp. 1064–1065 (cited on pages 20, 30).

[Grinter1999] Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. “The geography of coordination: dealing with distance in RnD work”. In: *Proceedings of the International ACM SIGGROUP conference on Supporting group work*. Phoenix, Arizona, United States: ACM, 1999, pp. 306–315 (cited on pages 2, 14).

[Guo2010] Philip J. Guo et al. “Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows”. In: *Proceedings of the 32th International Conference on Software Engineering*. Cape Town, South Africa, 2010, pp. 1074–1083 (cited on pages 165, 206, 208).

[Guo2011] P.J. Guo et al. “Not My Bug! and Other Reasons for Software Bug Report Reassignments”. In: *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW 2011)*. 2011, pp. 395–404 (cited on pages 168, 193, 206).

[Hambridge1995] S. Hambridge. *Netiquette Guidelines*. RFC 1855 (Informational). Oct. 1995 (cited on page 103).

[Hassan2004] Ahmed E. Hassan and Richard C. Holt. “Studying The Evolution of Software Systems Using Evolutionary Code Extractors”. In: *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE ’04)*. IEEE Computer Society, 2004, pp. 76–81 (cited on page 117).

[Hassan2006] Ahmed E. Hassan. “Mining Software Repositories to Assist Developers and Support Managers”. In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. 2006, pp. 339–342 (cited on page 15).

[Hassan2008] Ahmed E. Hassan. “The road ahead for Mining Software Repositories”. In: *Frontiers of Software Maintenance*. 2008 (cited on page 15).

[Hassan2009] Ahmed E. Hassan. “Predicting Faults Using the Complexity of Code Changes”. In: *Proceedings of the 31st International Conference on Software Engineering*. 2009 (cited on pages 15, 157, 158, 160, 163, 179, 185, 199, 205, 210).

[Hattori2008] Lile Hattori et al. “Mining Software Repositories for Software Change Impact Analysis: A Case Study”. In: *Brazilian Symposium on Databases*. 2008, pp. 210–223 (cited on page 15).

[Hecker1999] Frank Hecker. “Setting up shop: The business of open-source software”. In: *IEEE Software* 16.1 (Jan. 1999), pp. 45–51 (cited on pages 216, 253).

[Herbsleb2001] James D. Herbsleb et al. “An empirical study of global software development: distance and speed”. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE ’01)*. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 81–90 (cited on page 98).

[Herbsleb2003] James D. Herbsleb and Audris Mockus. “An empirical study of speed and communication in globally distributed software development”. In: *IEEE Transactions on Software Engineering* 29.6 (2003), pp. 481–494 (cited on pages 20, 35, 38, 40, 54, 55, 58, 60, 63, 65, 68–70, 72–75, 77).

[Herbsleb2006] James Herbsleb and Jeff Roberts. “Collaboration In Software Engineering Projects: A Theory Of Coordination”. In: *Proceedings of the 2006 International Conference on Information Systems* 38 (2006) (cited on pages 35, 36, 55, 57, 58, 60, 62, 63, 65, 66, 69, 70, 72, 74, 75).

[Herbsleb2008] James Herbsleb et al. “Socio-technical congruence (STC 2008)”. In: *Companion of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 1027–1028 (cited on page 14).

[Herraiz2006] Israel Herraiz et al. “The processes of joining in global distributed software projects”. In: *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner (GSD ’06)*. Shanghai, China: ACM, 2006, pp. 27–33 (cited on pages 98, 109).

[Hindle2009] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. “What’s hot and what’s not: Windowed developer topic analysis”. In: *Proeedings of the 25th IEEE International Conference on Software Maintenance* (2009), pp. 339–348 (cited on page 117).

[Hippel2003] Eric von Hippel and Georg von Krogh. “Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science”. In: *Organization Science* 14.2 (Mar. 2003), pp. 209–223 (cited on pages 226, 227).

[Hossain2009] Liaquat Hossain and David Zhu. “Social networks and coordination performance of distributed software development teams”. In: *Journal of High Technology Management Research* 20.1 (2009), pp. 52–61 (cited on pages 35, 37, 55, 58, 60, 63, 69, 70, 72, 74, 75).

[Hulkko2005] Hanna Hulkko and Pekka Abrahamsson. “A multiple case study on the impact of pair programming on product quality”. In: *Proceedings of the 27th International Conference on Software Engineering, ICSE 2005*. Saint Louis, MO, United states: Institute of Electrical and Electronics Engineers Computer Society, 2005, pp. 495–504 (cited on pages 35, 39, 40, 55, 58, 60, 63, 65, 66, 69, 71, 72, 74, 75).

[IBM2009] IBM. *Lotus Notes*. http://www.ibm.com/lotus/notes. Last visited March 2009. 2009 (cited on page 94).

[Jalali2012] Samireh Jalali and Claes Wohlin. “Systematic literature studies: database searches vs. backward snowballing”. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. Lund, Sweden: ACM, 2012, pp. 29–38 (cited on pages 25, 28, 29).

[JenkinsSmith2011] Hank C. Jenkins-Smith et al. “Enhancing acceptability and credibility of repository development for spent nuclear fuel”. English. In: vol. 1. Albuquerque, NM, United states, 2011, pp. 166–173 (cited on page 26).

[Jeong2009] G. Jeong, S. Kim, and T. Zimmermann. “Improving bug triage with bug tossing graphs”. In: *Proceedings of the 2009 ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2009)*. 2009, pp. 111–120 (cited on pages 168, 207).

[Jiang2008] Hsin-Yi Jiang et al. “Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management”. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE ’08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 59–68 (cited on pages 126, 130).

[Jiang2013] Yujuan Jiang, Bram Adams, and Daniel M. German. “Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. San Francisco, CA, USA: IEEE Press, 2013, pp. 101–110 (cited on pages 6, 85, 110, 230, 247, 252).

[Kamiya2002] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: a multilinguistic token-based code clone detection system for large scale source code”. In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670 (cited on page 137).

[Kim2006] Sunghun Kim, Kai Pan, and E. E. James Whitehead Jr. “Memories of bug fixes”. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT ’06/FSE-14)*. Portland, Oregon, USA: ACM, 2006, pp. 35–45 (cited on page 132).

[Kitchenham2009a] Barbara Kitchenham et al. “Systematic literature reviews in software engineering - A systematic literature review”. In: *Inf. Softw. Technol.* 51.1 (Jan. 2009), pp. 7–15 (cited on pages 20, 23, 30).

[Kitchenham2009b] B. Kitchenham et al. “The impact of limited search procedures for systematic literature reviews - A participant-observer case study”. In: *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*. 2009, pp. 336–345 (cited on page 27).

[Kolcz2004] Aleksander Kolcz, Abdur Chowdhury, and Joshua Alspector. “Improved robustness of signature-based near-replica detection via lexicon randomization”. In: *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD ’04)*. Seattle, WA, USA: ACM, 2004, pp. 605–610 (cited on page 109).

[Kraut1995] Robert E. Kraut and Lynn A. Streeter. “Coordination in Software Development”. In: *Commun. ACM* 38.3 (Mar. 1995), pp. 69–81 (cited on pages 2, 45).

[Krishnamurthy2005] Sandeep Krishnamurthy. “An Analysis of Open Source Business Models”. In: *Perspectives on Free and Open Source Software (Making Sense of the Bazaar)*. The MIT Press, 2005, pp. 279–296 (cited on pages 216, 253).

[Kutner2004] Michael H. Kutner, Christopher J. Nachtsheim, and John Neter. *Applied Linear Regression Models*. Fourth International. McGraw-Hill/Irwin, Sept. 2004 (cited on page 178).

[Kwan2011] Irwin Kwan, Adrian Schroter, and Daniela Damian. “Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project”. In: *IEEE Transactions on Software Engineering* 37.3 (May 2011), pp. 307–324 (cited on pages 14, 35, 52, 55, 58, 60, 61, 63, 64, 69, 72, 74, 75).

[Lee1999] Mong-Li Lee et al. “Cleansing Data for Mining and Warehousing”. In: *DEXA*. Ed. by Trevor J. M. Bench-Capon, Giovanni Soda, and A. Min Tjoa. Vol. 1677. Lecture Notes in Computer Science. Springer, 1999, pp. 751–760 (cited on page 96).

[Li2001] Shanjian Li and Katsuhiko Momoi. “A composite approach to language/encoding detection”. In: *Proceedings of the 19th International Unicode Conference*. 2001, p. 322 (cited on pages 98, 99).

[Lucia2007] Andrea De Lucia et al. “Recovering traceability links in software artifact management systems using information retrieval methods”. In: *ACM Trans. Softw. Eng. Methodol.* 16.4 (2007), p. 13 (cited on pages 126, 130).

[Maarek1991] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. “An Information Retrieval Approach for Automatically Constructing Software Libraries”. In: *IEEE Transactions on Software Engineering* 17.8 (1991), pp. 800–813 (cited on pages 126, 129).

[MacDonell2010] S. MacDonell et al. “How Reliable Are Systematic Reviews in Empirical Software Engineering?” English. In: *IEEE Transactions on Software Engineering* 36.5 (Sept.-Oct. 2010), pp. 676–87 (cited on pages 20, 28).

[Malik2008] Haroon Malik et al. “Understanding the rationale for updating a function’s comment”. In: *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM ’08)*. 2008, pp. 167–176 (cited on page 117).

[Marcus2003] Andrian Marcus and Jonathan I. Maletic. “Recovering documentation-to-source-code traceability links using latent semantic indexing”. In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 125–135 (cited on pages 117, 126, 130, 131).

[McCabe1976] Thomas J. McCabe. “A complexity measure”. In: *Proceedings of the 2nd International Conference on Software Engineering (ICSE ’76)*. San Francisco, California, United States: IEEE Computer Society Press, 1976, p. 407 (cited on pages 13, 157, 162).

[McConnell2004] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004 (cited on page 16).

[Meneely2008] Andrew Meneely et al. “Predicting failures with developer networks and social network analysis”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*. Atlanta, GA, United states: Association for Computing Machinery, 2008, pp. 13–23 (cited on pages 35, 45, 46, 55, 58, 60, 61, 63, 69, 72, 74, 75, 206).

[Meneely2009] Andrew Meneely and Laurie Williams. “Secure open source collaboration: An empirical study of Linus’ law”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS’09)*. Chicago, IL, United states: Association for Computing Machinery, 2009, pp. 453–462 (cited on pages 35, 47–49, 55, 58, 60, 63, 64, 69, 72, 74, 75).

[Meneely2010] Andrew Meneely and Laurie Williams. “Strengthening the empirical analysis of the relationship between Linus’ Law and software security”. In: *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM 2010)*. Bolzano-Bozen, Italy: Association for Computing Machinery, 2010, 9:1–9:10 (cited on pages 35, 49, 55, 57, 58, 60, 63, 64, 69, 70, 72, 74, 75).

[Mertsalov2009] K. Mertsalov, M. Magdon-Ismail, and M. Goldberg. “Models of Communication Dynamics for Simulation of Information Diffusion”. In: *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining (ASONAM’09)*. 2009, pp. 194–199 (cited on page 165).

[Microsoft2009a] Microsoft. *Microsoft Exchange*. http://www.microsoft.com/exchange. Last visited March 2009. 2009 (cited on page 94).

[Microsoft2009b] Microsoft. *Microsoft Outlook 2007*. http://www.microsoft.com/outlook/. Last visited March 2009. 2009 (cited on page 104).

[Mockus] Audris Mockus, Ping Zhang, and Paul Luo Li. “Predictors of customer perceived software quality”. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. Vol. 2005. Institute of Electrical and Electronics Engineers Computer Society, pp. 225–233 (cited on pages 19, 171, 205).

[Mockus2000a] Audris Mockus, Roy T. Fielding, and James Herbsleb. “A case study of open source software development: the Apache server”. In: *Proceedings of the 22nd international conference on Software engineering (ICSE ’00)*. Limerick, Ireland: ACM, 2000, pp. 263–272 (cited on page 229).

[Mockus2000b] Audris Mockus and Lawrence G. Votta. “Identifying reasons for software change using historic databases”. In: *ICSM ’00: Proceedings of the 16th IEEE International Conference on Software Maintenance*. 2000, pp. 120–130 (cited on page 117).

[Mockus2000c] Audris Mockus and David M. Weiss. “Predicting risk of software changes”. In: *Bell Labs Technical Journal* 5.2 (2000), pp. 169–180 (cited on pages 24, 35, 36, 53–56, 58, 60, 62, 63, 69, 70, 72, 74, 75).

[Mockus2002] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. “Two case studies of open source software development: Apache and Mozilla”. In: *ACM Trans. Softw. Eng. Methodol.* 11 (3 July 2002), pp. 309–346 (cited on page 254).

[Mockus2009] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. “Test Coverage and Post-verification Defects: A Multiple Case Study”. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 291–301 (cited on pages 196, 205).

[Monge2000] Alvaro E. Monge. “Matching Algorithms within a Duplicate Detection System”. In: *IEEE Data Eng. Bull.* 23.4 (2000), pp. 14–20 (cited on page 98).

[Moonen2001] Leon Moonen. “Generating Robust Parsers using Island Grammars”. In: *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE’01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 13–22 (cited on page 135).

[Moore1996] K. Moore. *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*. RFC 2047 (Draft Standard). Nov. 1996 (cited on pages 88, 100).

[Moser2008] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. In: *Proceedings of the 30th international conference on Software engineering (ICSE ’08)*. Leipzig, Germany: ACM, 2008, pp. 181–190 (cited on page 132).

[Munson1998] J. C. Munson and S. G. Elbaum. “Code Churn: A Measure for Estimating the Impact of Code Change”. In: *Proceedings of the International Conference on Software Maintenance (ICSM ’98)*. IEEE Computer Society, 1998, p. 24 (cited on pages 153, 157, 179).

[Nagappan2005] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density”. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE ’05)*. ACM, 2005, pp. 284–292 (cited on pages 13, 158, 179, 185, 205).

[Nagappan2006a] Nachiappan Nagappan, Thomas Ball, and Brendan Murphy. “Using Historical In-Process and Product Metrics for Early Estimation of Software Failures”. In: *Proceedings of the 2006 International Symposium on Software Reliability Engineering*. 2006, pp. 62–74 (cited on page 13).

[Nagappan2006b] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining Metrics to Predict Component Failures”. In: *Proceedings of the 28th International Conference on Software Engineering*. 2006, pp. 452–461 (cited on page 117).

[Nagappan2007] Nachiappan Nagappan and Thomas Ball. “Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study”. In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM ’07)*. IEEE Computer Society, 2007, pp. 364–373 (cited on pages 13, 157, 158, 170, 179, 199, 205, 210).

[Nagappan2008] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. “The influence of organizational structure on software quality: An empirical case study”. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*. Leipzig, Germany: Inst. of Elec. and Elec. Eng. Computer Society, 2008, pp. 521–530 (cited on pages 34–37, 41, 46, 54–56, 58, 60, 62, 63, 67, 69, 72, 74, 75).

[Navarro1999] Gonzalo Navarro and Ricardo Baeza-yates. “Very fast and simple approximate string matching”. In: *Information Processing Letters*. 1999, pp. 65–70 (cited on page 149).

[Nawaz2012] Ather Nawaz. “A Comparison of Card-sorting Analysis Methods”. In: *Proceedings of the 10th Asia Pacific Conference on Computer-Human Interaction (APCHI)*. 2012, pp. 583–592 (cited on page 32).

[Nguyen2008] T. Nguyen, T. Wolf, and D. Damian. “Global Software Development and Delay: Does Distance Still Matter?” In: *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2008)*. 2008, pp. 45–54 (cited on pages 35, 41–43, 55, 58, 60, 63, 65, 69, 72, 74, 75).

[Norton1987] John A Norton and Frank M Bass. “A diffusion theory model of adoption and substitution for successive generations of high-technology products”. In: *Management science* 33.9 (1987), pp. 1069–1086 (cited on page 154).

[Nurvitadhi2003] E. Nurvitadhi, Wing Wah Leung, and C. Cook. “Do class comments aid Java program understanding?” In: *Frontiers in Education, 2003. FIE 2003. 33rd Annual* 1 (2003), pages (cited on page 117).

[Ogasawara1996] Hideto Ogasawara, Atsushi Yamada, and Michiko Kojo. “Experiences of software quality management using metrics through the life-cycle”.

In: *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*. IEEE Computer Society, 1996, pp. 179–188 (cited on page 13).

[Ohlsson1996] Niclas Ohlsson and Hans Alberg. "Predicting Fault-Prone Software Modules in Telephone Switches". In: *IEEE Transactions on Software Engineering* 22.12 (1996), pp. 886–894 (cited on page 205).

[Oliveto2007] Rocco Oliveto et al. "Software Artefact Traceability: the Never-Ending Challenge". In: *Proceeedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE, 2007, pp. 485–488 (cited on page 126).

[Onyancha2012] I. Onyancha, A. Al-Awah, and F. Cole. "Addressing institutional potential loss of records and knowledge in Africa: The case of the ECA institutional repository - A knowledge base on African socio-economic development". English. In: *Int. Inf. Libr. Rev. (UK)* 44.3 (Sept. 2012), pp. 171–9 (cited on page 26).

[Overmeer2002] Mark Overmeer. "E-Mail Processing with Perl". In: *Proceedings of the third European Yet Another Perl Conference (YAPC 02)*. Munich, Germany: Munich Perl Mongers, 2002, pp. 34–44 (cited on page 95).

[Parnas1972] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Commun. ACM* 15 (12 Dec. 1972), pp. 1053–1058 (cited on pages 2, 45).

[Pattison2008] David Pattison, Christian Bird, and Premkumar Devanbu. "Talk and Work: a Preliminary Report". In: *Proceedings of the 2008 international workshop on Mining software repositories (MSR '08)*. Leipzig, Germany: ACM, 2008, pp. 113–116 (cited on page 14).

[Philips2000] Lawrence Philips. "The double metaphone search algorithm". In: *C/C++ Users J.* 18 (6 June 2000), pp. 38–43 (cited on page 119).

[Pinzger2008] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. "Can developer-module networks predict failures?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*. Atlanta, GA, United states: Association for Computing Machinery, 2008, pp. 2–12 (cited on pages 17, 18, 35, 46, 55, 58, 60, 62, 63, 69, 72, 74, 75, 206).

[PostgreSQL2009] PostgreSQL. *PostgreSQL Mailing List Archives*. http://archives.postgresql.org/. Last visited March 2009. 2009 (cited on page 92).

[Prause2012] C.R. Prause and Z. Durdik. “Architectural design and documentation: Waste in agile development?” In: *Proceedings of the 2012 International Conference on Software and System Process (ICSSP’12),* 2012, pp. 130–134 (cited on page 16).

[Pressman2001] Roger S. Pressman. *Software Engineering: A Practitioner’s Approach*. 5th. McGraw-Hill Higher Education, 2001 (cited on page 16).

[Purao2003] Sandeep Purao and Vijay Vaishnavi. “Product Metrics for Object-oriented Systems”. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 191–221 (cited on page 157).

[Ramasubbu2007] Narayan Ramasubbu and Rajesh Krishna Balan. “Globally distributed software development project performance: an empirical analysis”. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Dubrovnik, Croatia: ACM, 2007, pp. 125–134 (cited on pages 35, 40, 55–58, 60, 63, 69, 72, 74, 75).

[Ramasubbu2011] Narayan Ramasubbu et al. “Configuring global software teams: a multi-company analysis of project productivity, quality, and profits”. In: *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 261–270 (cited on pages 35, 43, 55, 58, 60, 63, 67, 69, 72, 74, 75).

[Rao2011] Shivani Rao and Avinash Kak. “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models”. In: *Proceeding of the 8th working conference on Mining software repositories*. ACM, 2011, pp. 43–52 (cited on page 144).

[Raymond1999] Eric S. Raymond. *The Cathedral and the Bazaar*. Ed. by Tim O’Reilly. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1999 (cited on page 126).

[Raymond2001] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Foreword By-Young, Bob. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2001 (cited on page 252).

[Rice1995] J.A. Rice. *Mathematical statistics and data analysis*. Statistics Series. Duxbury Press, 1995 (cited on page 249).

[Rigby2007] Peter C. Rigby and Ahmed E. Hassan. “What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List”. In: *Proceedings of the 4th International Workshop*

on Mining Software Repositories (MSR '07). Washington, DC, USA: IEEE Computer Society, 2007, p. 23 (cited on page 86).

[Rigby2008] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. "Open source software peer review practices: a case study of the apache server". In: *Proceedings of the 30th international conference on Software engineering (ICSE '08)*. Leipzig, Germany: ACM, 2008, pp. 541–550 (cited on pages 220, 229, 254).

[Robles2005] Gregorio Robles and Jesus M. Gonzalez-Barahona. "Developer identification methods for integrated data from various sources". In: *SIGSOFT Softw. Eng. Notes* 30.4 (2005), pp. 1–5 (cited on pages 107, 109).

[Robles2009] Gregorio Robles et al. "Tools for the Study of the Usual Data Sources found in Libre Software Projects". In: *International Journal of Open Source Software and Processes* 1.1 (2009), pp. 24–45 (cited on page 94).

[Rosenblatt1956] Murray Rosenblatt. "Remarks on Some Nonparametric Estimates of a Density Function". In: *The Annals of Mathematical Statistics* 27.3 (Sept. 1956), pp. 832–837 (cited on page 243).

[Schroter2006] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. "Predicting component failures at design time". In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. Rio de Janeiro, Brazil: ACM, 2006, pp. 18–27 (cited on pages 132, 157, 158, 174, 205).

[Schroter2010] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. "Do Stacktraces help Developers Fix Bugs?" In: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*. IEEE Computer Society, 2010, pp. 118–121 (cited on page 185).

[Sethanandha2010] Bhuricha Deen Sethanandha, Bart Massey, and William Jones. "Managing Open Source Contributions For Software Project Sustainability". In: *Proceedings of the 2010 Portland International Conference on Management of Engineering and Technology (PICMET 2010)*. Bangkok, Thailand: IEEE, July 2010, pp. 1–9 (cited on pages 227, 255).

[Shannon2001] C. E. Shannon. "A mathematical theory of communication". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 5.1 (2001), pp. 3–55 (cited on page 160).

[Shihab2009a] E. Shihab, Zhen Ming Jiang, and A.E. Hassan. “Studying the use of developer IRC meetings in open source projects”. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. Sept. 2009, pp. 147–156 (cited on page 86).

[Shihab2009b] Emad Shihab, Jiang Zhen Ming, and Ahmed E. Hassan. “On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK project”. In: *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. IEEE Computer Society, 2009, pp. 107–110 (cited on pages 86, 114, 117).

[Shihab2010a] Emad Shihab et al. “Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project”. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Bolzano-Bozen, Italy: ACM, 2010, 4:1–4:10 (cited on pages 178, 204, 205).

[Shihab2010b] E. Shihab et al. “Predicting Re-opened Bugs: A Case Study on the Eclipse Project”. In: *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE 2010)*. 2010, pp. 13–16 (cited on pages 168, 206).

[Shin2011] Yonghee Shin et al. “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities”. In: *IEEE Transactions on Software Engineering* 37.6 (2011), pp. 772–787 (cited on pages 35, 50, 55, 58, 60, 63, 64, 69, 72, 74, 75).

[Sim2003] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. “Using benchmarking to advance research: a challenge to software engineering”. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE ’03)*. Portland, Oregon: IEEE Computer Society, 2003, pp. 74–83 (cited on page 92).

[Sinclair1998] J.M. Sinclair. *Collins English Dictionary*. HarperCollins Publishers Limited, 1998 (cited on page 85).

[Sliwerski2005] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When do changes induce fixes?” In: *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR ’05)*. St. Louis, Missouri: ACM, 2005, pp. 1–5 (cited on pages 3, 117, 126, 131, 139, 157, 174, 207).

[Souza2004] Cleidson R. B. de Souza et al. “How a good software practice thwarts collaboration: the multiple roles of APIs in software development”. In: *SIGSOFT Softw. Eng. Notes* 29.6 (Oct. 2004), pp. 221–230 (cited on pages 2, 14).

[Spinellis2006] Diomidis Spinellis. “Global software development in the freeBSD project”. In: *Proceedings of the 2006 international workshop on Global software development for the practitioner*. Shanghai, China: ACM, 2006, pp. 73–79 (cited on pages 35, 39, 40, 55, 58, 60, 63, 66, 67, 69, 72, 74, 75).

[Steel1960] R.G.D. Steel and J.H. Torrie. *Principles and procedures of statistics: with special reference to the biological sciences*. McGraw-Hill, 1960 (cited on page 172).

[Strauss1990] A. Strauss and J. Corbin. *Basics of qualitative research: Grounded theory procedures and techniques*. Newbury Park, CA: Sage Publications, 1990 (cited on pages 219, 251).

[Tang2005] Jie Tang et al. “Email data cleaning”. In: *Proceedings of the 11th ACM SIGKDD international conference on Knowledge discovery in data mining (KDD ’05)*. Chicago, Illinois, USA: ACM, 2005, pp. 489–498 (cited on pages 86, 109).

[Tassey2002] Gregory Tassey. “The economic impacts of inadequate infrastructure for software testing”. In: *National Institute of Standards and Technology, RTI Project* 7007.011 (2002) (cited on page 153).

[Terceiro2010] A. Terceiro, L. R. Rios, and C. Chavez. “An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects”. In: *Software Engineering (SBES), 2010 Brazilian Symposium on*. 2010, pp. 21–29 (cited on pages 35, 49, 54, 55, 58, 60, 63, 65, 69, 72, 74, 75).

[Thomas2011] Stephen W Thomas. “Mining software repositories using topic models”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 1138–1139 (cited on pages 85, 110).

[Thorndike1920] Edward L. Thorndike. “A constant error in psychological ratings”. In: *Journal of Applied Psychology* 4 (1920), pp. 25–29 (cited on page 154).

[Vandecruys2008] Olivier Vandecruys et al. “Mining software repositories for comprehensible software fault prediction models”. In: *Journal of Systems and Software* 81 (5 2008), pp. 823–839 (cited on page 15).

[Wasserman1994] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences)*. 1st ed. Cambridge University Press, 1994 (cited on pages 165, 208).

[Wattenberg1999] Martin Wattenberg. “Visualizing the stock market”. In: *CHI ’99: CHI ’99 extended abstracts on Human factors in computing systems*. Pittsburgh, Pennsylvania: ACM, 1999, pp. 188–189 (cited on page 146).

[Wei2006] X. Wei and W. B Croft. “LDA-based document models for ad-hoc retrieval”. In: *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2006, pp. 178–185 (cited on page 145).

[Weissgerber2008] Peter Weissgerber, Daniel Neu, and Stephan Diehl. “Small patches get in!” In: *Proceedings of the 5th international working conference on Mining software repositories (MSR ’08)*. Leipzig, Germany: ACM, 2008, pp. 67–76 (cited on pages 86, 254).

[Weyuker2007] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. “Using Developer Information as a Factor for Fault Prediction”. In: *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, p. 8 (cited on pages 35–37, 55, 58, 60, 63, 69, 72, 74, 75).

[Whistler2008] Ken Whistler, Mark Davis, and Asmus Freytag. *The UNICODE Character Encoding Model*. Tech. rep. 17. The Unicode Consortium, Nov. 2008 (cited on page 98).

[Wnuk2009] Krzysztof Wnuk, Björn Regnell, and Lena Karlsson. “What happened to our features? visualization and understanding of scope change dynamics in a large-scale industrial setting”. In: *Proceedings of the 17th IEEE International Conference on Requirements Engineering (RE’09)*. IEEE. 2009, pp. 89–98 (cited on page 224).

[Wolf2009] Timo Wolf et al. “Predicting build failures using social network analysis on developer communication”. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE’09)*. Vancouver, BC, Canada: IEEE Computer Society, 2009, pp. 1–11 (cited on pages 2, 24, 35, 48, 49, 55, 58, 60, 63, 64, 69, 72, 74, 75, 157, 206).

[Wu2003] James Wu, T. C. N. Graham, and Paul W. Smith. “A Study of Collaboration in Software Design”. In: *Proceedings of the 2003 International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 304– (cited on pages 2, 85, 263).

[Wu2011] Rongxin Wu et al. “ReLink: recovering links between bugs and changes”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged, Hungary: ACM, 2011, pp. 15–25 (cited on page 149).

[Xie2007] Tao Xie, Jian Pei, and Ahmed E. Hassan. “Mining Software Engineering Data”. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 172–173 (cited on page 15).

[Yin2009] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009 (cited on pages 206, 248).

[Zeller2008] Andreas Zeller. “Learning from software”. In: *ISEC '08: Proceedings of the 1st conference on India software engineering conference* (2008) (cited on page 117).

[Zeller2009] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009 (cited on pages 157, 163).

[Zimmermann2004] T Zimmermann and P Weißgerber. “Preprocessing CVS data for fine-grained analysis”. In: *Proceedings of the International Workshop on Mining Software Repositories*. ACM, 2004, pp. 67–76 (cited on page 117).

[Zimmermann2005] T Zimmermann et al. “Mining version histories to guide software changes”. In: *Software Engineering* (2005) (cited on page 117).

[Zimmermann2007] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. “Predicting Defects for Eclipse”. In: *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*. Minneapolis, MN, USA, May 2007 (cited on pages 13, 15, 158, 202, 203, 205).

[Zimmermann2008a] Thomas Zimmermann and Nachiappan Nagappan. “Predicting defects using network analysis on dependency graphs”. In: *Proceedings of the 30th international conference on Software engineering (ICSE '08)*. Leipzig, Germany: ACM, 2008, pp. 531–540 (cited on pages 17, 205).

[Zimmermann2008b] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. “Predicting Bugs from History”. In: *Software Evolution*. Springer, Feb. 2008. Chap. Predicting Bugs from History, pp. 69–88 (cited on page 136).

[Zimmermann2009] Thomas Zimmermann et al. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process”. In: *Proceedings of the 2009 ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 2009, pp. 91–100 (cited on page 179).