Mining Unstructured Software Data

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Alberto Bacchelli

under the supervision of
Prof. Dr. Michele Lanza

June 2013

Dissertation Committee

Dissertation accepted on 14 June 2013

To Randy, the elephant in the room.

Abstract

The availability of large amounts of recorded data, produced during software development, has led to a research area called mining software repositories (MSR). Researchers mine software repositories both to support software understanding, development, and evolution, and to empirically validate novel ideas and techniques. Most MSR research focuses on mining archives of data that is either written by humans for a computer (*e.g.*, source code) or generated by a computer for humans (*e.g.*, execution traces). This data has an easily parseable structure that allows precise fact extraction and concerns the end product of software development. For this reason, the knowledge embedded in *structured data* can be extracted and modeled through well-established techniques.

Other software repositories archive data that is more unstructured, as it is produced by humans for humans: documents, such as emails, change comments, or bug reports, written in natural language and used to exchange information among people. The data contained in these repositories is not widely exploited because of its noisy and unstructured nature. The information stored in *unstructured data* encodes knowledge not to be found in other software artifacts, and also allows gaining valuable insights on the human factors revolving around a software project.

Our thesis is that the analysis of unstructured data supports software understanding and evolution analysis, and complements the data mined from structured sources. To this aim, we implemented the necessary toolset and investigated methods for exploring, exposing, and exploiting unstructured data.

To validate our thesis, we focused on development email data. We found two main challenges in using it to support program comprehension and software development: The disconnection between emails and code artifacts and the noisy and mixed-language nature of email content. We tackle these challenges proposing novel approaches. First, we devise lightweight techniques for linking email data to code artifacts. We use these techniques for creating a tool to support program comprehension with email data, and to create a new set of email based metrics to improve existing defect prediction approaches. Subsequently, we devise techniques for giving a structure to the content of email and we use this structure to conduct novel software analyses to support program comprehension.

In this dissertation we show that unstructured data, in the form of development emails, is a valuable addition to structured data and, if correctly mined, can be used successfully to support software engineering activities.

Acknowledgements

What made the years that led to my Ph.D. one of the best periods of my life are, no doubt, the people who I had the luck to meet and who decided to share some of their precious time with me. Thanks to them I grew, I had life changing experiences, I had fun, and I managed to successfully complete my Ph.D. with a happy smile on my face.

In my vision, a Ph.D. starts and ends with the advisor. I know for sure that there are great Ph.D. advisors out there (I have even seen some of them in action), but—for me—Prof. Michele Lanza has been nothing less than the perfect one. Michele, I will always be grateful to you for giving me the priceless chance to be one of your Ph.D. students. From you I have learnt so many things that I could write a book as long as this one. You have been a mentor for both my professional and my personal life. Among so many other positive things, you have *always* been be there for me when I needed, and you perfectly knew how to push me well beyond what I thought were my limits. And you probably did much more for me and all your Ph.D. students, than we could actually see. Thank you for everything. A special thank you to your sweet and phenomenal wife Marisa. Marisa, thanks for helping Michele to deal with us even during hard times!

My Ph.D. could not have been possible without the efforts and the approval of my dissertation committee: Prof. Lionel Briand, Prof. Fabio Crestani, Prof. Max Di Penta, Prof. Carlo Ghezzi, and Dr. Tom Zimmerman. I would like to thank you for accepting to be in my committee: I never expected to be so fortunate that top researchers like you decided to be part of my Ph.D. work. I thank you for all the valuable time you have spent on revising my progress and on providing me with top quality feedback. I really appreciated how you were, at the same time, professional, insightful, and easy-going. Lionel, thank you for accepting to evaluate my work despite your busy research agenda; I really appreciated your valuable feedback during the defense and your praises on the quality of my work. Fabio, thank you for letting me hear the voice of a top researcher in information retrieval about my work. Max you are one of the most hard-working, prolific, and smart people I had the chance to meet, thank you for all the constructive comments about my work; I am looking forward to start our research collaboration. Carlo, thank you for giving me good advices about my future career and for showing me how you can be both a great researcher and a cool person. Tom, thank you for giving me the opportunity to collaborate with you during my internship at Microsoft Research, I have learnt a lot from you and you have been of great inspiration during my research.

During my two internships at Microsoft Research, Dr. Christian Bird has been a great mentor, collaborator, and friend. Chris, thank you for asking me to work with you and for making my internships stellar from every perspective. You taught me a lot on how to do great research. It was so fun to learn together how to do a great qualitative study. I cannot forget how together we managed to submit an awesome ICSE paper the last day of my internship at 2 in the morning. Also, thank you for being a true mentor, who helped and understood me even about my personal issues. I am looking forward to continue collaborating with you!

I would like to thank all my colleagues in the REVEAL research group, led by Michele Lanza. Ricky, Mircea, Lile, Fernando, Marco, and Romain you made each of my working days special.

You were not only great colleagues, but also great friends from the very first day I arrived in Lugano. Ricky, thank you for making the office a funny place and for sharing with me a lot of your experience on work and life. Mircea, thank you for being the most welcoming person in the office, your very social nature was the perfect glue for all the relationships in the office. Lile, it has been great to work with you, even on very small projects, you showed me how one can be very professional, but at the same time not a workaholic. Fernando, every time I have to deal with object oriented issues, I *always* think about you and how you would solve them: You are a programming guru for me. Marco, you are the best colleague that I could have ever imagined; every day, I miss working with you on cool projects while having fun and making jokes. Romain, you were the best post-doc ever: As Michele has been the perfect advisor to me, you were the perfect post-doc, nothing less; one of the best things about continuing working in academia is that I can still collaborate with you.

I would like to thank Prof. Anthony Cleve and Dr. Andrea Mocci, who collaborated with me on a very cool piece of work. Not only it has been a pleasure to work with you, but also I found two great friends. I will continue to have fun and collaborate with you.

Many thanks to Prof. Mehdi Jazayeri, the founding dean of the University of Lugano, who gave me the opportunity to work there. Thank you for being such a great example of enlightened academic and educator. Also, thank you for your great help to improve my job applications to academia.

Concerning the University of Lugano, many thanks also to the nice people working at the "decanato." You helped me dealing with my traveling and logistic issues always with a cheerful smile and a friendly attitude, it was always a pleasure to go to your office.

Teaching is learning. During my Ph.D. years, I had the great opportunity to supervise many Bachelor and Master students. Ebrisa, Francesco, Luca, Lorenzo, Remo, Tommaso, and Vitezslav, I would like to thank you all for giving me the possibility to help you in achieving your targets and for letting me learn from you. A special thanks to Tommaso, who is also a great friend and decided to join the REVEAL group as a Ph.D. student after having worked with us for six months. Another special thanks to Sylvie, who I did not mentor directly, but who is always interested in my progress and who became a good friend.

Good luck and thank you to the new Ph.D. students in the REVEAL group: Roberto, Luca, and Tommaso. Be aware: You have to meet very high standards of coolness to maintain the former spirit of the group ;)

Another thanks goes to Dr. Roel Wuyts, who has been my teacher in the software engineering course during my Erasmus in Brussels. Roel, I would like to thank you because, undoubtedly, without that course I would have never decided to do a Ph.D. with Michele. So, thank you a lot! I hope to have the chance someday to thank you in person.

The last, but not least, "academic" thank you goes to Prof. Arie van Deursen, the SERG group, and TU Delft. I feel privileged that you decided–even before I defended my thesis–to give me the chance to continue my academic career in your great group, as an assistant professor! Thank you for all your support and your faith in me.

The first non-academic thank you goes to the outermost circle of my "family:" Alessandra, Gianluca, and Daniele. Thank you for being my best friends, for always being there for me, and for being such amazing people. Your friendship has been a great support for me during my Ph.D. years. Alessandra, your perseverance and love in your work is continuous inspiration for me: You know what it means to live for a passion. I am impressed about how, sometimes, you understand me even better than myself and how you can let me know what you think just by looking at me. Gianluca, I admire how you are able to find the best in the small things of life, and how you are so proud of your roots. Our friendship grew impressively in the last years, during which we had fun together in good times and you were with me on the bad ones, trying to make me think and cheer me up. I wish you and Alessandra all the best with your family. Daniele, you have been on my side since kindergarden, and I consider you as a brother. I know I will always be able to rely on you, and you know you can do the same with me. I wish you all the best with your new American adventure, from which you will be able to get out the most for you and for the people close to you. To all the three of you, thank you for being such an important part of my wedding: Gianluca and Daniele for being my official best men, and Alessandra for being my best "man" *in pectore*.

The second family thank you goes to my mom, Maria Cristina, and my sister, Chiara. Mom, you brought me to life and taught me to be a good person; you taught me what is important and what is valuable, you taught me what is right, and you showed me how to stand the troubles of life. Of course, no one can ever replace you. *Sorella*, I know I can rely on you for everything, because you have always been there for me. You trusted me more than everyone else and put complete confidence on me, since the very beginning. Thank you for being as you are. And thank you to Lucia, Michele, and Rita: The best nephews I could ever imagine.

I would like to say thank you to Oinky, Monky, Randy, Penguina, and Tommy. For being such an awesome *tribù* and for supporting me patiently while I was writing this thesis.

Finally, the last and most important thank you goes to the innermost circle of my family: My wife, Anja. Anja, every and each day of our life you make me discover what true love feels like. Thank you for choosing me as the companion for your life, for always exceeding my expectation under every possible aspect of everything, and for being the most important part of my life.

Alberto Bacchelli

Contents

List of Figures

List of Tables

Part I

Prologue

Chapter 1

Introduction

Human factors play a key role in software engineering processes. In 1968, Melvin Conway stated the hypothesis, later called *Conway's Law*, that any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure [48]. In the process of validating this law, recent research (*e.g.*, [38; 44; 143]) provides evidence that "*when the organization of people in a software effort is similar to the organization of the software, the project does better than when they differ*" [149]. In Fred Brook's seminal work *The Mythical Man-Month*, written in 1975, we read that the number of people involved in a software project increases its complexity exponentially. By analyzing the history of five hundred projects, DeMarco and Lister reported that in most failed projects "*there was not a single technological issue to explain the failure*" [124].

To understand human factors in software engineering, one should study the people involved in a software project as they work, typically by conducting *field research* [41]. Field research consists of "*a group of methods that can be used, individually or in combination, to understand different aspects of real world environments*" [117] and always requires data collection. Lethbridge *et al.* surveyed how field research has been performed in software engineering and accordingly proposed a taxonomy of data collection techniques by grouping them in three main sets [117]: (1) direct, (2) indirect, and (3) independent. Direct data collection techniques (*e.g.*, focus groups, interviews, questionnaires, or think-aloud sessions) require researchers to have direct involvement with the participant population; indirect techniques (*e.g.*, instrumenting systems, or "fly on the wall") require the researcher to have only indirect access to the participants' via direct access to their work environment; independent techniques (*e.g.*, analysis of tool use logs, documentation analysis) require researchers to access only work artifacts, such as issue reports or source code. Even though direct techniques allow the experimenter to obtain a general understanding of the human factors in the software engineering process and might be the only ones to gauge "*how enjoyable or motivating certain tools are to use or certain activities to perform*" [117], they present several drawbacks that threaten their validity and feasibility. For example, direct techniques are *observational* thus might suffer from the Hawthorne effect [79] (*i.e.*, the output of a process is not related to environmental conditions, but rather to whether or not subject are being observed); moreover, in direct techniques "*all data is potentially useful and the usefulness of a particular piece of data is often not known until after it is collected*" [175], thus the researcher could omit and not record significant details while conducting the observation; in addition, the data quality is often based on subjects' ability to remembering, which can be biased [175]; finally, finding appropriate subjects for a case study is hard, especially in industrial and distributed settings [201].

Given the drawbacks of direct data collection techniques, many researchers are performing investigations by means of independent data collection. In particular, software development tools such as version control systems, issue trackers, and mailing list services produce and record a large amount of information stored in software data repositories. By mining these software repositories, researchers extract data both to empirically validate novel research ideas and to support practitioners' day-to-day activities such as program comprehension, reverse engineering, or re-documentation tasks [92].

1.1 The Problem

The focus of most research in mining software repositories is directed to repositories containing *structured data*, such as code change repositories. The information in such repositories is well structured, because it comprises artifacts either written by humans for a machine (*e.g.*, source code, formal specifications and models) or generated by a machine for humans (*e.g.*, execution traces). The knowledge embedded in structured data can currently be extracted and modeled through well-established techniques. For this reason, most of the tools, achievements, and research are directed to the final product, the outcome, of software development, rather than the people and the process that generate the software product.

Structured data answers only little about human factors revolving around a software project. This is a problem because people and collaboration play a central role in software engineering [124]. In this vein, in 2009, at the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), Robert DeLine pointed out that only 20% of the papers in the 31st edition of the International Conference on Software Engineering (ICSE) dealt with software "*as though it were created by people working together.*"

To achieve results and answers that are closer to those of direct field research and to provide new perspectives and insights to understand and support software development, in our dissertation we want to mine *unstructured data* repositories, which are influenced by human factors. Such software data sources archive data that is produced by humans for humans: documents, such as emails, change comments, or bugs' reports, written in natural language and used to exchange information among people.

Unstructured data poses difficult challenges to the researchers who want to retrieve meaningful and relevant information. However, believing in the advantages that having this new form of information at our disposal would bring, we are convinced that "*we should not be dissuaded from our duty by the existence of textual narrative in artifacts*" [62]. In this dissertation, we want to investigate this form of information, with the aim of broadening our view and complementing current approaches based on structured data to improve system development and understanding.

1.2 Thesis Statement

We state our thesis as follows:

> *The content of unstructured data, such as emails, produced during the evolution of a software project is a valuable information source to support software understanding and evolution, and to complement the data mined from structured sources, such as source code artifacts.*

To validate our thesis, we mostly focused on development email data, although the presented approaches and findings can be adapted to other unstructured data artifacts, such as bug report comments and change comments. For the validation, we devised a number of approaches and implemented them in a comprehensive toolset to face the two main challenges in email and unstructured data mining:

Disconnection Between Unstructured Artifacts and Code Artifacts. Unstructured data, which is to be read both by the people involved in the evolution of the system and by the people that use it, often reference, explicitly or implicitly, other data sources such as source code and log reports. However, the actual linking to these entities is to be done by the reader. Moreover, the links are one way: There is no link from source code to unstructured data or vice-versa. Connecting unstructured data to the source code is a necessary step to use this data in the context of software development and analysis.

Noisy and Mixed-language Content Unstructured data is often noisy form of information: It is not formal, contains irrelevant data, information might be wrong or incomplete, and authors are not professional writers and often use jargon. Unstructured data related to a software system, then, do not only contain sentences written in natural language, but also fragments written in other "languages," such as source code, execution traces, or patches. To extract *relevant* and *correct* information from unstructured data, it is necessary to remove the unwanted data, recognize the different languages, and subsequently give a structure to the content (thus enabling techniques to exploit the peculiarities of each language).

The validation of our thesis consists in devising and applying analysis techniques on top of our toolset. With these techniques we show that, by analyzing unstructured content, we support software understanding and evolution.

1.3 Contributions

The main contributions of this dissertation can be classified in four categories: exploratory investigation, tools and mining approaches, analyses, and publicly available benchmarks.

Exploratory Investigation

1. We conducted an iterative exploratory investigation to disclose challenges and benefits of exploiting email data to support understanding and development of software projects. This is the methodology we follow in our dissertation (see Section 3.1).
2. We identified the importance of reconnecting development emails to source code artifacts (see Chapter 4).
3. We realized the importance of *identifying* the structure in email data (see Chapter 7);
4. We identified the importance of *understanding* the structured content in email data (see Chapter 8). We realized this cannot be achieved using a lexical approach and regular expressions, but it requires a full-fledged parsing approach.
5. We identified that development emails are composed of a number of languages (see Chapter 9) that should be recognized to enable subsequent ad-hoc analyses.
6. We conducted a qualitative study [86] to understand what data can be found in OSS mailing lists (see Chapter 10). We conduct this research to guide future investigations on this form of unstructured software data.

Tools and Mining Approaches

7. We present MILER [14; 15] an extensive and extensible meta-model and toolset to explore email and unstructured data (see Section 3.2).
8. We devised, implemented, and tested a number of lightweight approaches [13] to recover the traceability links between emails and source code artifacts (see Section 4.3).
9. We conducted a comprehensive evaluation of state-of-the-art linking techniques [18]. We compare different linking methods, ranging from our lightweight approaches to more complex approaches from the information retrieval (IR) field (see Section 4.5).
10. We extended MILER to model bug information. We extend our meta-model to support new metrics generated by our lightweight email-to-code linking approach [12] (see Chapter 5).
11. We created REMAIL [16; 17; 9], an Eclipse plugin based on our email-to-code linking approach. It makes email data available in the IDE, *i.e.*, the place where developers spend most of their time (see Section 6.2).
12. We devised and evaluated lightweight techniques that detect source code fragments in emails by exploiting characteristics of source code text (see Chapter 7).
13. We devised ILANDER [10], an island parser to recognize, extract, and model source code fragments immersed in natural language text. ILANDER is based on the ASF-SDF Meta-Environment [196], and we evaluate it extracting information from emails (see Chapter 8).
14. We created PETITISLAND, a *flexible* and *extensible* framework for building and composing island parsers. PETITISLAND is written in SMALLTALK and is based on the parser generator PETITPARSER[163]. We evaluated it by extracting source code fragments from Stack Overflow posts (see Chapter 8).
15. We created a novel approach that fuses island parsing and machine learning techniques for classification of email lines [11]. Our approach, named MUCCA, is able to perform automatic classification of the content of development emails into five language categories: natural language text, source code fragments, stack traces, code patches, and noise (see Chapter 9).
16. We created a novel web application to manually classify email content. We extend the MILER GAME by creating a novel user interface and adding new features (see Chapter 9).
17. We created a coding system that is reusable for analysis of developer communication in general, and mailing lists in particular (see Chapter 10).

Analyses

18. We show that a number of program comprehension tasks are enhanced by having email data at disposal (see Chapter 6). We used REMAIL (see Section 6.2) and the connection between emails and source code to complete this task [17].
19. We devised a novel defect prediction technique based on email data. We evaluated the performance of our model on four OSS systems [12].

20. We use iLander to conduct a number of software evolution analyses. We reconstruct the model of a software system from its emails and we detect salient moments in its history. (see Chapter 8).
21. We assessed the frequency of discussion topics in development mailing list (see Chapter 10). In particular, we found that implementation details are not extremely prominent.

Publicly Available Benchmarks

22. We produced two benchmarks for evaluating the recovery of traceability links between emails and source code artifacts (see Chapter 4). We created them by analyzing the mailing lists of six diverse OSS systems written in four different programming languages. It includes more than 5,000 manually annotated emails.
23. We produced a benchmark that features sets of sample emails, randomly extracted from five unrelated Java OSS systems, which we manually read to label structured fragments. It includes more than 1,800 manually labelled emails (see Chapter 7).
24. We produced a benchmark for evaluating the recognition, extraction, and modeling of Java content in email data (see Chapter 8). We created it by reading and labeling sample emails from four unrelated Java OSS systems. It comprises 188 labelled emails with described structured fragments.
25. We adapted and improved a previously published benchmark to assess the recognition of source code fragments in Stack Overflow posts. It comprises 188 posts embedding more than 350 code fragments (see Chapter 8).
26. We produced a benchmark to evaluate the classification of email lines into five categories, *i.e.*, natural language, source code, patch, stack trace, and noise (see Chapter 9). It features more than 1,400 emails comprising almost 69,000 manually classified lines.
27. We create two benchmarks: one for email thread categorization and one for resolving aliases of participants (see Chapter 10). Our benchmark are comprised of more than 500 manually classified threads and more than 310 resolved aliases and email addresses.

1.4 Outline

The rest of the dissertation is structured as follows:

Part I: Prologue. In the first part of this dissertation, we introduce the background and the motivation for our thesis. We present the challenges to prove our thesis and the methodology we followed in our work.

Chapter 2 provides a historical perspective on our work, by presenting the history of mining software repositories and analyzing the (more recent) work on using unstructured data for software engineering. We also survey work related to the analysis of email data.

Chapter 3 presents the methodology we used in the course of our dissertation, describing our iterative exploratory research, the toolset we devised for supporting our research, and the decision taken in terms of case studies and replication.

Part II: Linking Unstructured Data and Source Code Artifacts. In this part of our dissertation, we cover the first challenge of mining development email data: Recovering the traceability links between emails and source code artifacts. We devised lightweight traceability techniques and evaluate them against state-of-the-art IR techniques. Building on top of our lightweight techniques, we present a novel defect prediction analysis and a tool, which we used to evaluate the usefulness of email data to support program comprehension.

Chapter 4 presents our lightweight traceability techniques, based on simple text matching, and the evaluation we conducted against state-of-the-art IR techniques. We surprisingly found that, in the case of emails, simple text matching still offers the best results. Reasoning on the results we realized that, differently from other natural language documents, which are written *before* the actual implementation, emails are being sent while developers and users are creating and using the actual application. For this reason, most of the time, code artifacts that already exist are referred to using their actual names. As a consequence our lightweight techniques offer the most effective results.

Chapter 5 presents a defect prediction analysis based on email data. Having the traceability links between source code entities and emails, we devised a set of new metrics to enrich a system model with information extracted from email archives. Such metrics seize the "popularity" of source code entities in the discussions taking place in emails. We used these new metrics to perform defect prediction for object-oriented systems at class level, and we compared their predictive power to that of metrics obtained through structured data (*i.e.*, object-oriented metrics, change/defect metrics). We achieved results similar to source code metrics, but inferior to change metrics. The most interesting contribution of our metrics is that, by combining the metrics extracted from repositories with different form of data (*i.e.*, structured and unstructured), we could improve the overall predictive power.

Chapter 6 introduces REmail. By using our lightweight traceability techniques, which provide results in a few seconds even when linking one code entity to thousands of emails, we implemented REmail, an Eclipse plugin, to make email data available in the place where developers spend most of their time–IDEs. We subsequently used REmail to verify that having email data at disposal in the development environment enhances tasks related to program comprehension and software development.

Part III: Structuring Unstructured Data. In this part of our dissertation, we cover the second challenge of mining development email data: Giving an appropriate structure to the unstructured content. To this aim, we first tried lightweight methods for detecting code content. Then, realizing that detection is not sufficient, we created an approach, based on island parsing, to recognize, extract, and model structured fragments in development emails. This granted us a better understanding of email content. We present a number of analyses that used this novel information. We improved our approach by implementing a flexible and extensible island parsing framework, and using it on unstructured data other than development emails: Stack Overflow posts. Finally we show how this approach could be used to support other software development tasks.

Chapter 7 presents our initial work on trying to give a structure to the unstructured email data. We found that emails often contain structured fragments, such as source code, which should not be treated equally to the rest of the content. We devised lightweight techniques that, on the basis of simple text inspections, exploiting characteristics of source code text, can detect source code fragments in emails, fast and with a high accuracy. Our methods achieved performance higher than the ones previously obtained through complex machine learning techniques.

Chapter 8 presents iLander, an island parsing approach based on ASF-SDF. After evaluating it against a manually implemented benchmark, we used it to conduct software analysis on an OSS system. Afterwards, we present PetitIsland, a more comprehensive, extensible, and flexible framework we devised to create and compose island parsers.

Chapter 9 presents Mucca, an approach to classify development email lines. Given the diversity of languages used in the example email, if we consider its content as a single bag of words, we would obtain a motley set of flattened terms without a clear context, and we would severely reduce the quality and the amount of available information. By automatically distinguishing the parts that form an email, we provide better support for subsequent analyses and tasks, such as traceability recovery or content summarization. Mucca fuses island parsing and machine learning to propose a robust and accurate approach.

Part VI: Epilogue In the last part of our dissertation we take a step back from the different challenges and analyses done on development emails and consider our work as a whole and we look further to find the directions for future research. Since we sensed that mailing lists in OSS communities have been facing a shift with respect to their usage, we first conducted a work to update our knowledge on the data available nowadays in mailing list. This helps us to target our future research to the most important information available in mailing list and also to the most promising repositories of unstructured data. Subsequently, we present our contributions and outline future research directions we envision.

Chapter 10 presents an extensive qualitative research on the development mailing list of a mature and widely used OSS system. Our aim is to update our knowledge of mailing list usage and data. We found that development emails are losing their role as the hub of project communication and that other unstructured data sources are gaining more popularity, such as issue repositories.

Chapter 11 concludes this dissertation by discussing our approaches and findings, summarizing the contributions of this work and outlining future research directions.

Chapter 2

State of the Art

In our dissertation we propose approaches for mining unstructured software data, and we apply these approaches to real data to show that this form of data is useful to support software understanding and evolution. This is challenging because unstructured data is not written with software evolution analysis in mind. For example, development mailing lists can contain the discussions behind certain implementation choices, but extracting this information and connecting it to the appropriate code artifacts is not trivial. In this chapter, we analyze the background for our research and the current state of the art in mining unstructured software data.

2.1 A Historical Perspective

In this section, we describe the progress in research that led to the birth of the mining software repositories (MSR) research field. The emergence MSR goes hand in hand with the recognition of software development and maintenance as an evolutionary process, named *software evolution*.

The Seventies: SCM Applied to Software. In the seventies Software Configuration Management (SCM) emerged as a discipline. In 1975, the same year when the first International Conference on Software Engineering (ICSE) was held, Rochkind introduced the first SCM, called Source Code Control System (SCCS) [170]. Although in the fifties the aerospace industry already started a sort of configuration management, SCCS was the first case in which configuration management was applied to software.

In 1976, Mills argued that software development should be incremental with continuous user participation [132]. In 1977 Gilb proposed evolutionary project management, and introduced the terms *evolution* and *evolutionary* to the lexicon of the software process [78]. In 1978, Yau introduced evolutionary elements in software development process models [206]. In 1979, Feldman developed the program *Make*, making an important contribution to SCMs and their adoption.

The Eighties: Software Evolution As a Discipline. Manny Lehman introduced the 'laws of software evolution', which describe a set of general principles for the evolution of *E-type* software systems. This kind of software systems is one that "*mechanise a human or societal activity*" [115] and needs to change to adapt to the real world and maintain its usefulness. The formulated empirical laws were based on a study to understand the change process being applied to IBM's OS 360 operating system. Lehman confirmed the software evolution laws on other software systems in

1985 [116]. Lehman used the term *software evolution* to emphasize the difference with the post-deployment activity of software maintenance. It took until the end of the eighties for the term software evolution to be widely accepted [6; 148].

During the eighties, SCM systems continued to mature and increased their adoption: In 1982, Tichy introduced Revision Control System (RCS) [191], while four years later Concurrent Version System (CVS), a very popular versioning system (still used in a large number of OSS projects [72] and industrial settings) emerged.

The Nineties: Widespread Usage of SCM. In the nineties, the concepts of software evolution and evolutionary development became widespread. In the same decade, SCM received a significant acceleration in attention and usage. With the advent of the Internet and the improvement in network bandwidth, software development started to be distributed, source code repositories started to be remote and CVS played a key role in this transition, as it supports concurrent development. Moreover, in the nineties, the first bug tracking systems were created: GNATS being the first in 1992, followed by Debbugs in 1994, Bugzilla in 1998, a service offered by SourceForge in 1999 and many others after 2000 (including the Google Code issue tracker in 2007).

At the end of the nineties and in the following decade two important events in the growth of SCM were the creation of Subversion (SVN) in 2000 (the successor of CVS) and the release of Git in 2006 (an open source distributed version control system).

2.2 Mining Software Repositories

The Dawn of Mining Software Repositories. In the second half of the nineties, researchers started to mine repositories of source code data. The first approaches were proposed by Ball *et al.* in 1997 to find clusters of files frequently changed together [19], by Graves *et al.* in 1998 to compute the effort necessary for developers to make changes [84] and by Atkins *et al.* in 1999 to evaluate the impact of tools on software quality [8]. These are among the seminal research works where the field of mining software repositories has its roots.

Starting from the first half of the last decade, the usage of SCM systems became fundamental for software development. Estublier *et al.* stated: "[...] *modern SCM systems are now unanimously considered to be essential to the success of any software development project* [...]. *Furthermore, there is a lively international research community working on SCM, and a billion dollar commercial industry has developed*" [68]. A number of new bug tracking systems were created (*e.g.*, Jira) and their usage started to become established practice in software development.

The availability of large amounts of data on the evolution of software systems drew the attention of researchers and practitioners to software repositories, to a point that mining software repositories matured and started to be a research area on its own. In 2004, the first International Workshop on Mining Software Repositories (MSR) was held [95]. In the following years, the topic gained increasing attention and the field continued to mature: Many software engineering and software maintenance conferences had sessions about mining and the international workshop on MSR became a working conference in 2008 [92].

The Mining Software Repositories Field. The official website of the conference describes the MSR field: "[The MSR] *field analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects.*"[1] The main goal of MSR is to make intelligent use of software repositories to support decision making, to empirically validate novel research ideas, and to support practitioners' day-to-day activities such as program comprehension, reverse engineering, or re-documentation tasks. Software repositories offer a rich source of data, but it must be correctly processed, transformed, and presented to be useful.

Most software development tools, such as version control systems, issue trackers, and mailing list services, produce and record a large amount of information, which is the result of traces left by the evolutionary changes to software artifacts (*e.g.*, source code), and the interaction among the stakeholders and project members. We describe the most prominent sources of data:

Source code. The source code is a set of instructions to be executed by a computer and written using some human-readable computer language, usually as text. The source code repository contains a collection of documents, written in one or more programming languages (*e.g.*, JAVA) and usually grouped in packages or modules.

SCM data. The software configuration management system (also known as versioning system or revision control) collects data by recording the history of changes made to a set of documents. Documents can be added, deleted, and changed. Usually software developers use versioning systems to store the edits to the source code of the projects they are working on. This allows them, for example, to retrieve previous versions of a system.

Execution logs. The execution logs record the output of a software system during its execution. The output to be logged by an execution is generally defined by developers by means of programming languages and might contain, for example, the name of methods being called and the time of execution.

Issue repositories. When developers or users of a system encounter an error or a faulty behavior, they create an *issue report* in the issue tracking system. Reports are usually discussed, assigned to developers, and resolved. More recently issue repositories also archive enhancement requests, in the form of new features requests or improvements.

Requirements and Design Documents. Requirements are documents, mostly written in natural language, that establish the needs of stakeholders to be solved by the software system. The requirements can be either *functional* (to specify the behavior of the program) or *non-functional* (to define qualities such as reliability and accessibility). Design documents describe the general design of a software system, including its architecture and use cases.

Mailing list logs. Mailing lists archive messages exchanged among participants in a software project. In OSS systems, mailing lists are considered to be the hub of project communication [73], and are employed to discuss to discuss various topics, ranging from low-level concerns (*e.g.*, bug fixes, refactoring) to high-level resolutions (*e.g.*, future planning).

Chat logs. Chat logs are the recorded instant messaging communication among participants in a software project. Usually chat logs include, for all messages, authors and timestamps.

Online forums. Online resources, such as forums and question and answers (Q&A) services, provide developers with the infrastructure to exchange knowledge in form of questions and answers: Developers pose questions and receive answers regarding issues from people that are not part of the same project.

1 `http://msrconf.org`

2.2.1 Linking Software Data Sources

A problem in the exploitation of the data recorded in software repositories is their integration. Usually projects use issue tracking systems, mailing list managers, and version control systems that are not connected one to another. This means, for example, that there is no link connecting software defects recorded in an issue tracking system to the source code artifacts they affect. To conduct successful mining of software repositories, recovering links among repositories is essential [52], particularly reconnecting non-code repositories (*e.g.*, issue repositories) to source code artifacts they pertain to. To deal with this issue, researchers proposed many techniques; D'Ambros categorized them in: *during development* approaches and *after development* ones [52]:

During development These approaches enhance software development tools (*e.g.*, integrated development environments (IDE)) to support the linking across multiple repositories and to show relevant artifacts to the users. An example is Jazz [75], which integrates source code with defects, defects with failed unit tests, source code with builds and builds with defects. Another example of this type of approach is Mylyn [104; 103].

After development These approaches reconnect data in software repositories by analyzing the contained data itself. Most of the approaches regard the linking between SCM repositories and bug databases: The first approach was the Release History Database (RHDB) [69], followed by others such as Kenyon [100], Hipikat [194] and softChange [77].

In our case, there is no possibility to use *during development* approaches. In fact, unstructured data, such as development emails, offers no actual linking to referenced artifacts, and there is no link from code to unstructured data. Recovering the links between unstructured data and software artifacts, so that the former can be more effectively used, is the first challenge we have to face to prove our thesis. In Part II of this dissertation we tackle this challenge and show the subsequent benefits for supporting software understanding and evolution analysis.

2.2.2 Structured and Unstructured Software Data

By mining software repositories, researchers have produced valuable results. The focus has mainly been on repositories of *structured data*, which is easier to extract data from. In fact, structured data has a well defined meta-model and a known form, because it is either designed to be automatically parsed by a machine (*e.g.*, in the case of source code, test plans, and code changes) or it is reported by a computer (*e.g.*, execution logs). The main sources of structured data currently being analyzed by researchers are:

- **Source code:** Source code has been studied, analyzed, and measured to assess, for instance, the quality of the internal structure of software systems (*e.g.*, [128; 113]) or to predict the location of software defects in future releases (*e.g.*, [21; 183; 142; 210; 53]).
- **SCM systems**: SCMs have attracted the interest of researchers more recently [19; 213] and are being studied to understand the evolution of a system [80; 54], to identify components that present design flaws [161], to support new debugging methods [208], or to improve existing defect prediction techniques (*e.g.*, [152; 137]).
- **Issue repositories**: Issue repositories are being mined to investigate the defects in the history of a software system: Researchers devised defect prediction techniques studying historical defects (*e.g.*, [94; 27; 105]), or performed retrospective system analysis to understand the most problematic parts of a system [55].

The other information stored in software repositories is *unstructured data*. Most repositories contain a mixture of structured and unstructured data. To define the term unstructured data for this dissertation, similarly to Thomas [188], we adopt the following definition [123]:

> "*Unstructured data is data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records.*"

In practice, unstructured data usually is natural language text, which follows no explicit data meta-model, and it is written by people for other people. One potential criticism to this definition is that that also natural language text has a structure, made of relationships among words, an almost linear flow, and terms with a well defined syntactic or morphological behavior. Although this is true, because such a structure can be more or less easily inferred by humans, in the case of automatic analysis, such a parsing is far from being simple and actionable in the current status of the research. For this reason, in its raw form, "[natural language] *text is simply a collection of characters with no structure and no meaning to a data mining algorithm*" [188]. Examples of unstructured software data, archived in software repositories, are: emails exchanged by program participants, the text of chat logs, titles and descriptions of issue reports, source code comments and identifiers, SCM commit messages, and design and requirement documents.

Structured data is not good for everything. There are some kinds of questions for which structured data answers little: Especially in the context of understanding and exploiting human factors to understand and support software development, structured data limits us to partial views. Let us imagine two developers working in the same project: We parse code changes to determine whether they worked on the same files in the same time period; we parse email metadata, to verify whether they also communicated in these days; but we do know whether they were *really* collaborating, whether they were aware of each other work, and whether they had according plans. To have answers, we should query them by conducting a field study with *direct* data collection, with the subsequent drawbacks (see Chapter 1). Otherwise, we could perform *indirect* data collection and study the *content* of their discussions or commit comments to get deeper insights. In fact, people write such documents to share *knowledge* with other people, thus they contain facts and qualitative data that answer new kinds of questions.

Our thesis stems from our belief that natural language documents–if correctly mined, measured, and made available–can integrate, consolidate, and complement the data extracted from structured sources, because they include human factors and are a source of qualitative data.

The repositories that store artifacts with textual narrative, however, are still largely unexplored by researchers and not exploited by software developers. On the one hand practitioners do not employ textual artifacts during development for many reasons [200]: Developers do not know whether a specific topic is expressed in these artifacts, fast full-text search is not always implemented, different kinds of artifacts provide different search and browsing tools, there is no consistency among different repositories and tools, and it is hard to know whether an artifact contains updated information or not. On the other hand, researchers must still find appropriate techniques for extracting relevant information from unstructured data.

Finding appropriate techniques for extracting relevant information from unstructured data is the second challenge in proving our thesis. In Part III, we tackle it by creating methods for extracting structured content embedded in emails, by creating an approach to give a structure to unstructured documents, and by showing that extracted information can be used to conduct new software analyses.

2.2.3 Emails as an Unstructured Data Source of Information

To prove our thesis, among all the possible unstructured data sources of information, we decided to focus on development emails, because they offer an interesting and hard case (due to their noisy and mixed language nature) to test novel mining techniques and conduct new software analyses. In the following we better describe the rationale of our choice and the current state of the art on mining this form of data.

In small co-located development teams, unplanned face- to-face meetings are the favorite form of communication when developers face program comprehension problems [114]. Developers who need to understand source code entities (*e.g.*, to know the design rationale behind a certain implementation–the most common information need for a developer [109]), and cannot find the appropriate documentation, simply query other programmers. This solution, besides disrupting developers' attention and retaining knowledge by a few developers, is inapplicable to large or distributed development projects. Developers, thus, replace face-to-face meetings with electronic communication. Instant messaging, wikis, forums are viable options, but the decisive role is played by emails, indeed: "*Mailing lists are the bread and butter of project communications*" [73].

Emails are asynchronous, thus evade time zone barriers and do not disrupt developers' attention; mailing lists broadcast discussions, announcement, and decisions to all the participants, thus maintaining developers' awareness; emails are not bound to specific abstraction levels (as opposed to commit messages, design documents, or code comments), thus they can be used to discuss issues ranging from low-level decisions (*e.g.*, implementation, bug fixing) up to high-level considerations (*e.g.*, design rationales): The range of topics of a discussion on a mailing list "*is larger than a bug database or CVS commit logs*" [166]; mailing lists archive messages during the whole lifetime of a software project, thus offering a historical perspective. From a field research point of view, email discussions are appealing because "*software developers reveal their thought processes most naturally when communicating with other software developers, so this communication offers the best opportunity for a researcher to observe the development process*" [175].

By investigating email archives, we can conduct two kinds of analyses: *social analyses* and *technical analyses*. The former focuses on studying and interpreting the social phenomena, within a particular software project, or in the context of *software ecosystems* [121], using approaches equivalent to that used by historians or social scientists [155]; the latter extracts the data enclosed in mailing lists to tackle technical issues and extend system analysis.

For these reasons, we focus our research on mining development email archives.

Email metadata offers an easily parsable structured content with information about author, date and time, and threading of email messages. Researchers used this structured information to conduct the first social analyses by employing emails: Bird *et al.* proposed techniques to mine email social networks [36], to discover that there is a strong cumulative relationship between email activity and source code activity [37], and to investigate social interaction among participants of open source software projects [38]. Ogawa *et al.* used email metadata to visualize social interaction among participants in open source software projects [146]. Tang *et al.* proposed techniques for identifying the country origin of participants in open source mailing lists, and conducted a subsequent geographic analysis [186]. Shihab *et al.* performed an exploratory study on the role of mailing lists in open source projects, and further showed that mailing list activity is related to source code activity; in addition, by starting to look into the content of emails, they found that specific words in mailing list discussions are good indicators of the types of source code changes happening in the project [178].

More recently researchers have started analyzing the natural language *content* of emails: Pattison *et al.* investigated the behavior of developers and users by studying the frequency with which terms of software entities are mentioned in emails, and correlating it with the number of system changes [154]. Baysal and Malton tried to correlate discussion in emails and source code [23]. In particular, they searched for a correlation between discussions and software releases, by applying data mining and Natural Language Processing techniques.

The current ongoing work and interest shown by researchers on development emails is a symptom of the acknowledged importance of this source of people-centric information and is paving the way for deeper and more detailed analysis of this data. Our thesis moves in this direction: We explored email data, as a case of unstructured data, to see the challenges that must be faced to expose the most important information (thus creating the appropriate mining techniques) and to realize how we can exploit this information to support program comprehension and software evolution (thus suggesting improved or novel software analyses).

2.3 Summary

The current ongoing work on mining software repositories and the increasing interest shown by practitioners and researchers on this field is a symptom of the acknowledged importance of this kind of research. The main focus in these years has been on structured data, because it comprises both the end product of software development (*i.e.*, source code) and it is easier to parse and to extract data from. Structured data is complemented by a large amount of unstructured data, usually in the form of natural language communication among project participants. This type of data is under exploited by the current state of the art in mining software repositories. Through our survey on the related work and our iterative exploration of unstructured data, with a focus on development emails, we found that the two main challenges in exploiting unstructured data for software engineering are reconnecting unstructured data artifacts and source code artifacts and extracting meaningful and actionable information from the mixed language and noisy content of unstructured artifacts. These are the two challenges we face in our dissertation. In the next chapters we will also detail the work related to each of our approach and analyses.

Chapter 3

Methodology, Replicability, and Subject Systems

Believing that tools have a fundamental role for software engineering research, we gradually developed a toolset Miler, which we used to test theories, conduct empirical studies, and perform software quality analyses. We carried out our research in an iterative explorative fashion; we iteratively: (1) investigated which information can be provided by email data, (2) found the problems that hinder the correct exploitation of such information, (3) devised and evaluate approaches to tackle the problem and implement them in our toolset to support the corresponding novel techniques and analyses, (4) reasoned about the results to gain a better knowledge of email data and the available useful information it contains, to use it in the next iteration.

3.1 Iterative Explorative Approach

We started our research by trying to create a software system visualization that contained information about which entities of the system were discussed in the development mailing list. This simple task made us aware of the difficulty in finding an appropriately evaluated method for linking emails and source code entities. We found no work in literature on how to link emails and source code and we realized that this link is the first necessary step of any work that is willing to use email data for software analysis. In fact, the majority of MSR approaches start from the model of the software system as it is extracted by the source code, and enriches it with additional information gathered from different sources, such as the versioning system or the issue repository. In this way, it is possible to obtain grounded and actionable results to support software understanding and development.

First Iteration: Reconnecting Emails and Code Artifacts. In the first iteration, we faced our first challenge: Recovering the traceability links between emails and source code artifacts. We started tackling it by building Miler, our toolset to support email data exploration (see Section 3.2). We used Miler to devise and evaluate lightweight lexical text-matching techniques for linking email and source code entities (see Chapter 4). Afterwards, we expanded our toolset to support more programming languages and we compared our lightweight linking techniques to more sophisticated information retrieval methods, such as vector space modeling (VSM). We found that our lightweight approaches are better suited for recovering traceability links, especially when dealing with code and emails.

Second Iteration: Using the Link Between Emails and Code Artifacts. In the second iteration, we exploited our lightweight techniques for reconnecting email data to source code artifacts. First, we went into the direction of adding new quantitative metrics to enrich our knowledge of source code entities with email information. In particular, we measured the *popularity* of code artifacts in mailing list discussions and used this information to conduct bug prediction analysis (see Chapter 5). We found that email data can be successfully used to enrich existing defect prediction models: It adds information that increases the models' explanative and predictive power. With this result we made a step toward proving our thesis.

Even though email data proved to be valuable for defect prediction, we realized that we did not know why this was the case: We were creating a measure that merely counted how much an entity was discussed without knowing *why* or *how* this was happening. This made us realize that the value of emails resides in the context they always provide: When a code artifact is mentioned in a message, it is referred in a context of a broader message that conveys more information. This reflection led us to the third iteration.

Third Iteration: Providing Email Data in The Development Environment. In the third iteration, we carried on our research on using the linking information, but we also went into the direction of taking advantage of the content of emails. We created REmail, an Eclipse plugin to integrate email communication in the IDE (see Section 6.2). By using REmail, developers can seamlessly handle source code entities and emails concerning them in the environment they use to develop. By using REmail we made a further step to prove our thesis: We successfully used it to show that email data can enhance our understanding of software development, by means of the contextual information it provides (see Section 6.3).

While studying email data for program comprehension, we realized that emails are composed of different languages: natural language, source code, stack traces, *etc.* We found that the kind of language in which the reference to a code entity appears significantly affects the usefulness of the email for different program comprehension tasks. For example, when debugging an entity, emails that reference it in a stack trace are more useful than emails that reference it in a code snippet, while when trying to understand the evolution of a code artifact, emails that reference it in a patch are the most relevant.

Fourth Iteration: Source Code Identification. In the fourth iteration, we moved our research on studying the *content* of emails and trying to recognize the structured content that they include, so that we can better extract further information. We created lightweight methods to find lines of source code, patches, or stack traces (see Chapter 7), in the body of an email. By studying the results of this research, we realized that we need not only a finer granularity (*i.e.*, distinguish code snippets from patches from stack traces) but also to recognize irrelevant parts of the emails' contents (*e.g.*, authors' signatures) and filter them out from our analysis.

Fifth Iteration: Parsing Structured Content in Unstructured Data. In the fifth iteration, we devised techniques (see Chapter 8) for precisely recognizing, extracting, and modeling certain structured parts embedded in any type of textual artifact. In fact, by only detecting structured lines we cannot understand their meaning and use it for subsequent analyses. We approached this problem by using island parsing [136], first by implementing an island parser in ASF-SDF (see Section 8.3). With this implementation we managed to conduct a first step into novel software analyses. Afterwards, given the difficulty of extend, maintain, and use our first solution based on ASF-SDF, we created a novel extensible framework for island parsing in Smalltalk (see Section 8.7). We used our framework to write island parsers for source code, patches, stack traces, and we used it to classify email lines (see Chapter 9).

Sixth Iteration: Updating Our Knowledge on Mailing List Usage Historically, mailing lists have been considered the hub of project communication at the inception of the first OSS communities, such as Linux and Apache. We, thus, consider mailing list data to be a valuable case study for devising and testing techniques for mining and analyzing unstructured data from software projects.

Nevertheless, in the course of the years, while developing and analyzing mailing lists, we sensed a change in the usage of development mailing lists across OSS systems. For this reason, to conclude our work, we decided to analyze carefully, with a qualitative research, the content of the development mailing list of a OSS project, across its entire history. We found that indeed the role of the development mailing list changed, by diminishing its importance in OSS project communication. At the same time, another source of unstructured data is emerging more and more: The issue repository. Software developers and users are increasingly using this type of repositories to interact, set new milestones, and discuss about the details of the project. Luckily, the techniques we devised to deal with mailing list data can be adapted to issue repository data, which is less noisy and more structured than email messages.

3.2 Tool Based Research

To import, process, store, and analyze both email and source code data, during our research, we devised the toolset Miler, by progressively and iteratively implementing its features.

Miler is implemented in VisualWorks Smalltalk, relies on the Moose Reengineering Environment [145] for some modeling tasks, and uses GLORP (Generic Lightweight Object-Relational Persistence) [107] and the Metabase (a tool to provide flexibility and persistence to any metamodel in general) [56] for the object persistency. Figure 3.1 depicts Miler's architecture: It shows the sources from which Miler retrieves data, the data importers, the external components being used, the kernel, the PetitIsland framework, and the processing/analysis modules.

3.2.1 A Walk Through Miler

Importing email data: Importing email data to be used as a source of information is not trivial, because there is no consistent way to access data. Different personal email clients and different applications to manage mailing lists store data in different formats. Moreover, in the life of a developer or a software system, the email client or the mailing list applications can be changed multiple times.

Accessing email data in this scenario would require writing multiple email data importers for each software system being analyzed. We tackled this issue by (a) creating an importer for the MBox format, which is used by many email clients and mailing list managers (*e.g.*, GNU Mailman), and (b) using MarkMail,[1] an online service for searching among more than 8,000 up-to-date mailing lists. Our importers either crawl the MarkMail website or parse the MBox

[1] `http://markmail.org/`

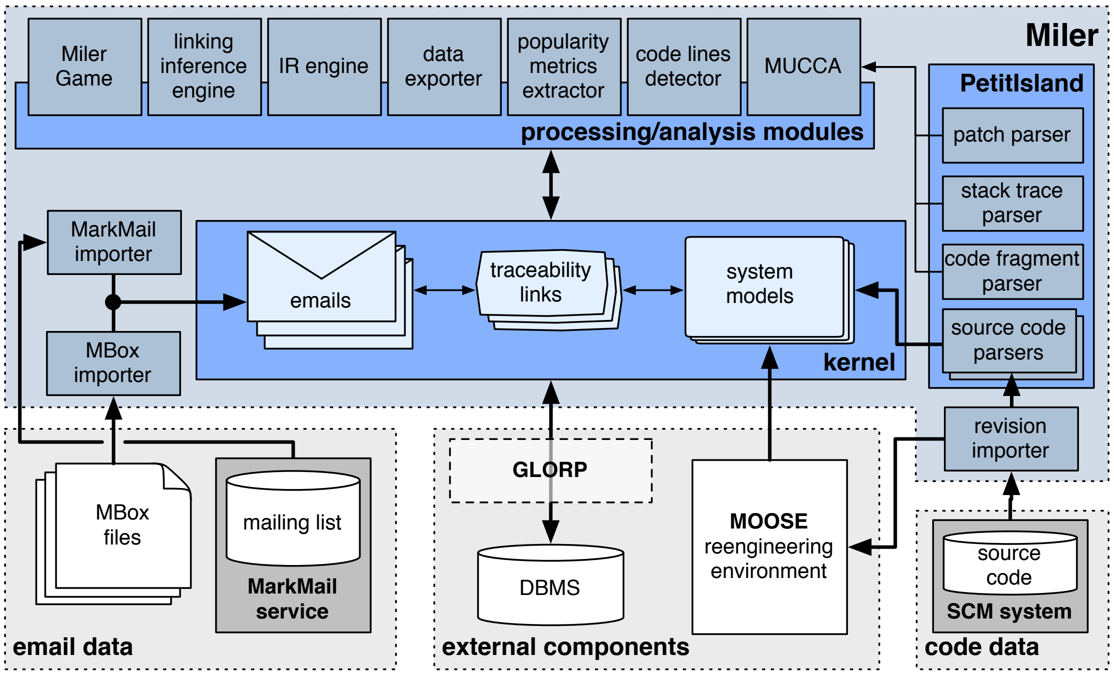


Figure 3.1: The architecture of Miler

file, extract the selected emails, and instantiate them as live objects in the Miler kernel. As shown in the architecture diagram, we can plug importers that extract data from additional sources.

Importing code data: Miler handles software systems written in many programming languages through specialized importers. First, we wrote a *revision importer* to import multiple releases of the same software system. It obtains multiple source code snapshots of a software system wither by downloading them from the system website (not depicted in Figure 3.1) or by performing "check out" operations (one per snapshot) from the software configuration management (SCM) system, also known as versioning system repository. In the case of retrieving the data from the versioning system, we can specify the timespan to be considered between subsequent releases.

Once Miler obtains the source code snapshots, we import them with two different techniques. For Java systems we have two options (which we used in different moments in time to perform different analyses): (a) We use inFusion[2] for parsing the source code and importing it in the Moose reengineering environment, so that we can extract a number of code metrics (*e.g.*, see Chapter 5); (b) we use a lightweight specialized Java code parser written on top of our parsing framework (see Section 8.7). The latter option is also used to import systems written in other languages: We devised lightweight island parsers for ActionScript, PHP, and C systems.

The models created by inFusion, and enriched by Moose, or created through our island source code parsers are imported in the Miler kernel as *System Models*.

The meta-model: Figure 3.2 shows the meta-model behind the Miler's kernel. `System` is the class representing any software system imported in Miler. Each `System` has a collection of `Re-`

2 `http://www.intooitus.com/products/infusion`

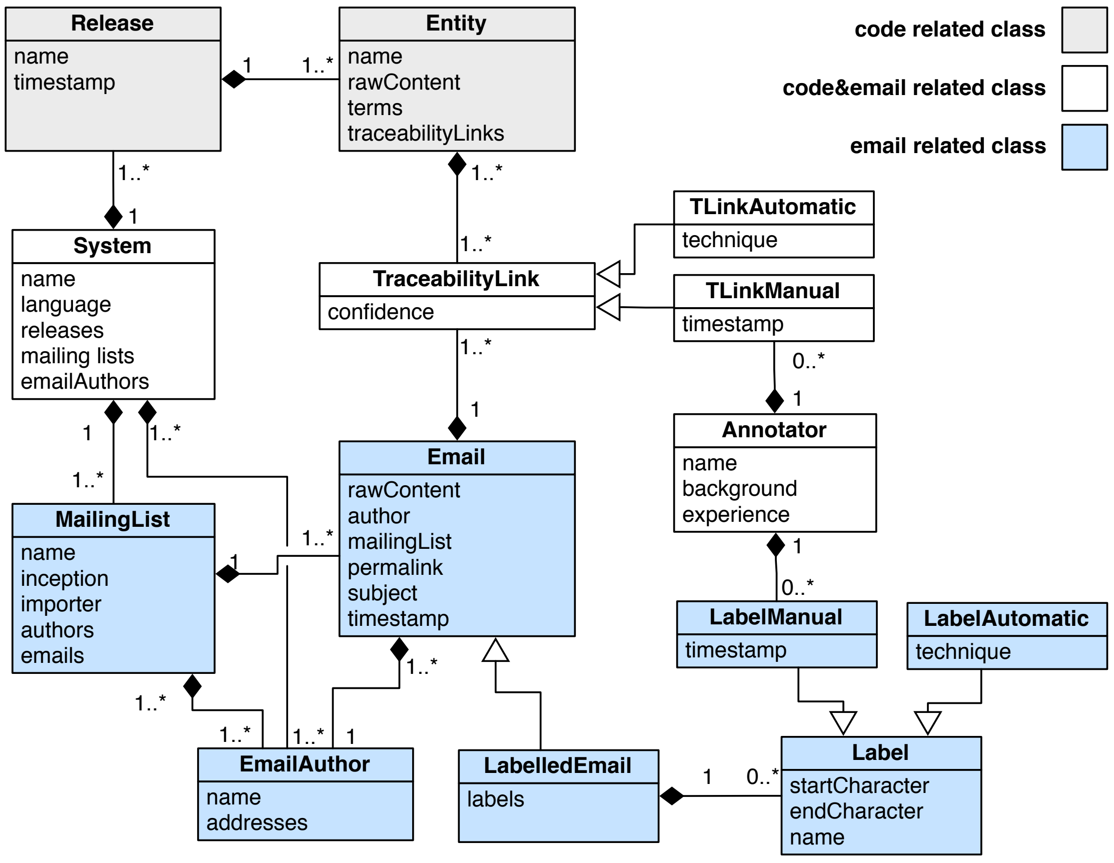


Figure 3.2: The Meta-Model of Miler

lease s, which represents the various snapshots of the source code. Each `Release` is characterized by a "timestamp" and has a collection of unique instance of the class `Entity`. In our case, even though we work in SMALLTALK and we could extend the meta-model definitions of MOOSE, we decided to create a new class to represent entities of the system. This allowed us to be independent from the MOOSE meta-model, which has been under heavy re-development and evolution during the course of our thesis work. Nevertheless, it is possible to substitute this abstraction with a `FAMIXAbstractObject` class.

Each `System` has also a collection of mailing lists (for example the same software system can have a user mailing list to complement the development one). Mailing lists are represented by the `MailingList` class, which has a list of email authors and a list of unique emails, described by the class `Email`. In addition to the raw content, each email has fields for storing meta-data information, such as the timestamp or the subject.

The *data exporter* module exports the imported and processed information as it is described by the meta-model in Figure 3.2, so that it can be used by other researchers and applications.

Storing imported data: To store information gathered from mailing lists and source code, we use an approach based on object persistency rather than using text files. Although the MOOSE environment uses textual files (in particular in the MSE format) to store and retrieve information about any analyzed system, we deemed them to be not appropriate in our context. In fact, despite the fact that textual files do not require a DBMS, they have the drawback that data cannot be accessed remotely, and that they generate performance bottlenecks, since the entire text file must always be parsed (*i.e.*, it is not possible to import only parts of the model). When considering mailing lists, the performance aspect is relevant as they often contain thousands of documents.

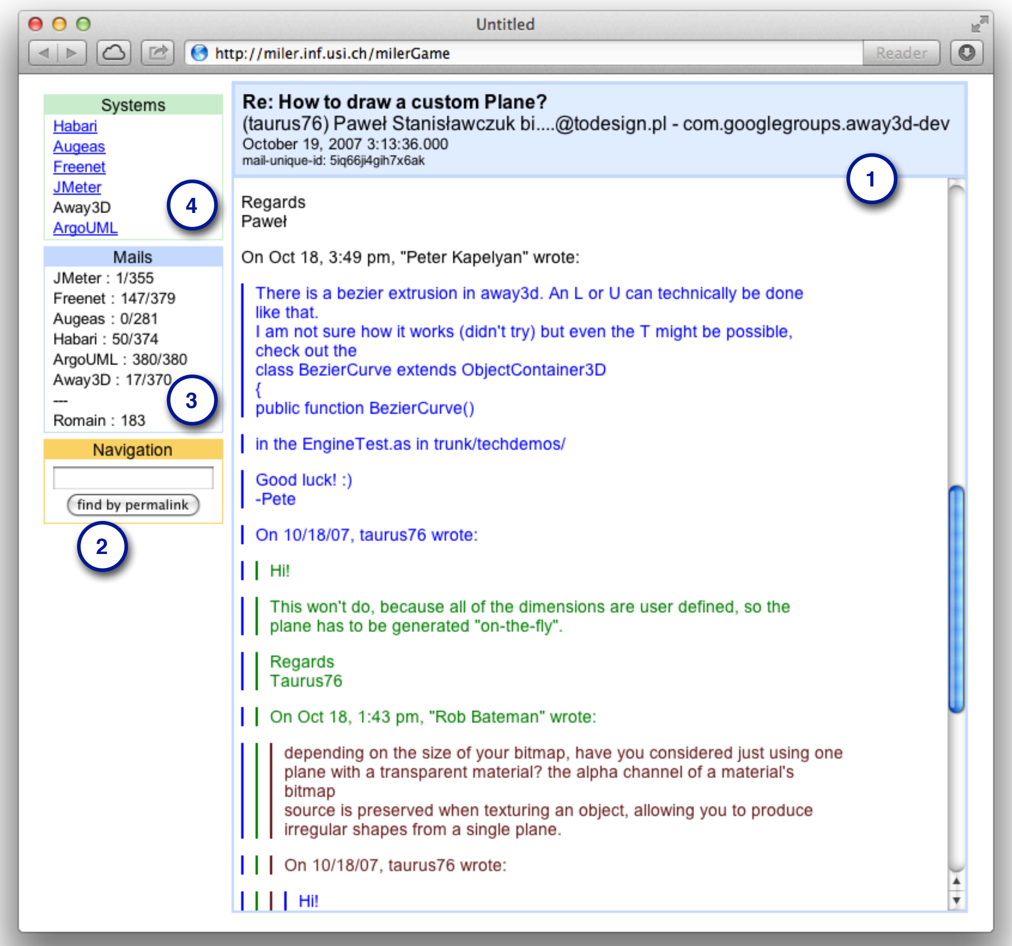
In Figure 3.1, the components that reside in the Miler's kernel are modeled according to a meta-model (depicted in Figure 3.2), from which the Metabase component is capable of automatically generating the corresponding GLORP class descriptions (which define the mapping between the SMALLTALK classes and the database tables) [56]. In this way, objects are stored and retrieved from the chosen database transparently through the GLORP layer: It is sufficient to save the objects of the model the first time they are created and to create a connection with the database when loading MILER. In addition, since objects are stored in a common database, it is possible to access them, even remotely, from different languages and applications.

Interacting with emails: Using the SMALLTALK web framework SEASIDE [63], we implemented the MILER GAME, a web module to interact with emails stored in the MILER kernel. Figure 3.3 shows the main interface of the MILER GAME.

The MILER GAME has a modular structure organized in panels. Point 1 marks the `main panel`, which displays the selected email, with meta-data on top and the whole body colored according to quotation levels to enhance readability. Within the MILER GAME, one can plug any number of panels to interact with the stored systems and emails. For example, on the left of the main panel, we see: the `navigation panel` (Point 2), to read a specific email, given its id; the `mails panel` (Point 3), with statistics on emails; and the `systems panel` (Point 4), to switch to another system.

Linking code and emails: Even though development emails often discuss source code artifacts, establishing actual links to the referenced entities is to be manually established by the reader. Moreover, the links are unidirectional: There is no visible link from source code to emails. This is the first challenge we encountered in our thesis (see Section 1.2). In the first phase of our research (see Chapter 4), we implemented a `linking inference engine` to automatically infer these traceability links and persist them in the MILER kernel. These links are first class entities in our meta-model, and are modeled by the class `TraceabilityLink`. The `TraceabilityLink` entities can be instantiated as `TLinkAutomatic` by the engine. In this case, the engine can use either lightweight text-matching techniques we devised, or the information retrieval (IR) techniques implemented in our IR engine (*e.g.*, LSI and tf-idf). With different case studies, we verified whether one of the techniques better handles this linking task.

Extracting metrics: Once email and source code data are imported, and email messages are reconnected to code entities they discuss, we can devise new metrics to measure new facts. To this aim, we devised the *popularity metrics extractor*, which enriches a system model with information extracted from email archives. In particular, the current implementation of the extractor computes metrics to seize the "popularity" of source code entities in the discussions taking place in emails. In mailing lists, the entities that are discussed are not only the most relevant for the development, but also those that are most exploited during the usage of the software system. Moreover, the email content is expressed using natural language, which does not require the writer to carefully explain all the abstractions using the same level of importance. Our extractor can be extended to include other metrics measuring different aspects extracted from email data.



JMeter : 1/355
Freenet : 147/379
Augeas : 0/281
Habari : 50/374
ArgoUML : 380/380
Away3D : 17/370

Romain : 183

Re: How to draw a custom Plane?
(taurus76) Paweł Stanisławczuk bi....@todesign.pl - com.googlegroups.away3d-dev
October 19, 2007 3:13:36.000
mail-unique-id: 5iq66ji4gih7x6ak

```
Regards
Paweł

On Oct 18, 3:49 pm, "Peter Kapelyan" wrote:

| There is a bezier extrusion in away3d. An L or U can technically be done
| like that.
| I am not sure how it works (didn't try) but even the T might be possible,
| check out the
| class BezierCurve extends ObjectContainer3D
| {
| public function BezierCurve()

| in the EngineTest.as in trunk/techdemos/

| Good luck! :)
| -Pete

| On 10/18/07, taurus76 wrote:

| | Hi!

| | This won't do, because all of the dimensions are user defined, so the
| | plane has to be generated "on-the-fly".

| | Regards
| | Taurus76

| | On Oct 18, 1:43 pm, "Rob Bateman" wrote:

| | | depending on the size of your bitmap, have you considered just using one
| | | plane with a transparent material? the alpha channel of a material's
| | | bitmap
| | | source is preserved when texturing an object, allowing you to produce
| | | irregular shapes from a single plane.

| | | On 10/18/07, taurus76 wrote:

| | | | Hi!
```

Figure 3.3: The Miler Game

Detecting structured fragments: The second challenge we faced in our thesis is the noisy and mixed-language content of development emails (see Section 1.2). In fact, development emails are often interleaved with structured content, *e.g.*, stack traces, patches, or code snippets, and noise such as author's signatures. We first tried to separate structured fragments from natural language in email messages. This brings several benefits, such as better characterization of mailing list usage, improved authors' behavior analysis, or reconstruction of alternative system models (see Chapter 7). Miler offers a *Structured fragments detector* module, which implements a lightweight technique we devised for classifying emails and lines containing structured data. With a case study, we evaluated the effectiveness of a number of candidate techniques for detecting structured fragments in email content.

A framework for island parsing in unstructured content: After we devised lightweight approaches for separating structured content from natural language in emails, we realized that much information is available in these fragments. For extracting facts from this information, the `structured fragment detector` module is not enough, because it is not able to parse and model the content. For this reason, we devised a more powerful approach based on island parsing, which is able to identify, extract, and model structured information embedded in textual data artifacts of any kind (*e.g.*, source code or development emails). This approach is implemented in a framework, called PetitIsland, based on the Smalltalk parser generator PetitParser [163]. Our framework is designed to be accurate and efficient, flexible, and extensible (see Section 8.7). We extended it with specialized island parsers for code fragments, patches, stack traces.

Line classification of development emails: Mucca (eMail Unified Content Classification Approach) is the last application built in Miler. It consists of a technique to classify email lines in five categories: natural language, source code, stack traces, patches, and noise. Mucca is based both on island parsers written on top of PetitIsland and on two machine learning techniques (*i.e.*, Naïve Bayes and Classification Tree). The machine learning techniques are implemented in Weka [90], which we use as an external component (not depicted in Figure 3.1) to have access to a vast number of machine learning approaches. With a case study, we evaluated the effectiveness of Mucca and found that it reaches almost perfect results (see Chapter 9).

3.3 Benchmark and Replicability

To mine unstructured data, researchers have been experimenting with technologies adopted from related research fields, such as topic models from Information Retrieval (IR), hierarchical clustering from Data Mining (DM), or part-of-speech tagging from Natural Language Processing (NLP). By studying these approaches (more background on the different IR approaches is given in the chapters in which they are evaluated or employed), we noted that there is a gap between the validation approaches used by the software engineering researchers and the ones used in IR, DM, and NLP. The areas in which IR, DM, and NLP techniques have proven useful (*e.g.*, management of scientific and legal literature, web searches) are supported by a set of well designed, robust, and universally accepted benchmarks. IR benchmarks, for example, are publicly available and distributed via the infrastructure of the Text REtrieval Conference series (TREC), sponsored by the National Institute of Standards and Technology (NIST) and the US Department of Defense (DARPA) [179]. They keep evolving and now include retrieval tasks for many different kinds of information (*e.g.*, spam, genomic data).

In short, work in related research fields is supported by an extensive and statistically significant benchmark against which their techniques were evaluated. This does not hold for the software engineering approaches.

Sim *et al.* were the first to report this trend in software engineering [179]. They stressed the importance of widely used and reliable benchmarks to assess the quality of research findings, and challenged the software engineering research community to define appropriate benchmarks. We share the concerns raised by Sim *et al.* For this reason, we support our work with carefully designed and publicly available benchmarks. This allows us to better generalize our findings and to let other researchers compare their approaches to ours and devise improvements.

In the analyses we conducted in which we needed benchmarks (*i.e.*, when we had to compare the output of automatic approaches to annotations done manually by human reviewers), we created such oracles manually by reading and annotating a large number of emails. For example, to establish the quality of the traceability links retrieved by our lightweight approaches, we needed a dataset in which emails were already linked to the correct code artifacts. Since it did not exist, we had to create it manually by reading thousands of emails. We aided these kinds of manual tasks by creating specialized panels implemented in the MILER GAME.

We implemented the MILER GAME to have the following features:

- It is web based, thus it can be accessed by different platforms and with different devices.
- It is easy to interact with it, because it shows only a single window with all the panels immediately accessible.
- It supports concurrency: More users can access the MILER GAME and concurrently make changes in the email data (*e.g.*, to add annotations).
- Changes and annotations done by the users are immediately stored and persisted in the database.
- It is extensible by means of pluggable panels.

Using extensions to the MILER GAME allowed us to make the task easier and less fatiguing, and to automatically include the results in our models. For example, manually retrieved traceability links are instances of the class `TLinkManual`. The data exporter module in MILER is useful for sharing our benchmarks and allows our work to be replicated. We refer the reader to the following chapters (*i.e.*, Chapter 4, Chapter 7, and Chapter 9), to see how we extended the MILER GAME to create different benchmarks for different scenarios.

3.4 Subject Systems for the Analyses

To conduct our data exploration and assess our techniques and findings, we performed a number of case studies. To have a great variety of data and projects to study, and to have the possibility to share the datasets we use in our research to improve verifiability and reproducibility of our results, we used open source software (OSS) systems for our case studies.

Table 3.1 summarizes the main characteristic of the OSS systems used in our case studies. In general, we selected mature systems with a mature development process, that make use of mailing lists as communication channels, and that have an established user base. In the following we briefly describe them to show that they vary in domain, usage, and development community:

ArgoUML is a graphical application for designing and automatically generating UML diagrams. It is written in JAVA and released under the open source Eclipse Public License.[3]

Augeas is a configuration-editing library. It parses configuration files in their native formats and transforms them into a tree. Manipulating this tree and saving it back into native configuration files make configuration changes. It is written in C and released as OSS under the GNU Lesser General Public License[4] (LGPL).

Away3D is a 3D graphics engine, which can be used to render 3D models and perform various other 3D computations. It is written for the Adobe Flash[5] platform in ACTIONSCRIPT 3 (an

3 `http://www.eclipse.org/legal/epl-v10.html`
4 `http://www.gnu.org/copyleft/lesser.html`
5 `http://www.adobe.com/software/flash/about/`

Chapters	System	Website	Programming Language	Initial Available Release	Development Mailing List	
					Name	Inception
4, 7, 8, 9	Argouml	http://argouml.tigris.org/	Java	Apr 1999	org.tigris.argouml.dev	Jan 2000
4	Augeas	http://augeas.net/	C	Dec 2007	com.redhat.augeas-devel	Feb 2008
4	Away3d	http://away3d.com/	ActionScript	Mar 2007	com.googlegroups.away3d-dev	May 2007
5	Equinox	http://www.eclipse.org/equinox/	Java	Feb 2003	org.eclipse.equinox-dev	Feb 2003
4, 6, 7, 8, 9	Freenet	https://freenetproject.org/	Java	Mar 2000	org.freenetproject.devl	Apr 2000
4	Habari	http://www.habariproject.org/	PHP	Apr 2007	com.googlegroups.habari-dev	Oct 2006
5	Jackrabbit	http://jackrabbit.apache.org/	Java	Sep 2004	org.apache.jackrabbit.dev	Sep 2004
4, 7, 9	JMeter	http://jmeter.apache.org/	Java	Mar 2001	org.apache.jakarta.jmeter-dev	Feb 2001
5, 10	Lucene	http://lucene.apache.org/	Java	Sep 2001	org.apache.lucene.java-dev	Sep 2001
5	Maven	http://maven.apache.org/	Java	Sep 2003	org.apache.maven.dev	Nov 2002
6, 7, 8, 9	Mina	http://mina.apache.org/	Java	Nov 2006	org.apache.mina.dev	Jan 2006
7	OpenJPA	http://openjpa.apache.org/	Java	Aug 2007	org.apache.openjpa.dev	May 2006

Table 3.1: The open source software systems used in our analyses, by chapters

object-oriented programming language compliant with the ECMAScript Language Specification[6]), and it is released under the Apache License 2.0.[7]

Equinox is a plugin system for the Eclipse project. It is an implementation of the Open Services Gateway initiative (OSGi) core framework specification,[8] a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. It is written in Java and released under the Eclipse Public License.

Freenet is a peer-to-peer software for anonymous file sharing, and for browsing and publishing "freesites" (web sites accessible only through Freenet). It is a platform for censorship-resistant communication. It is written in Java and released under the GNU General Public License[9] (GPL).

Habari is a blog engine, publishing platform, and application framework. It is written in object-oriented PHP 5, and supports many OSS database management systems for the database backend. It is released under the Apache License 2.0.

Apache Jackrabbit is a content repository is a hierarchical content store with support for structured and unstructured content, full text search, versioning, transactions, observation, *etc.* It is written in Java and released under the Apache License 2.0.

Apache JMeter is a load-testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications. JMeter architecture is based on plugins. It is written in Java and it is released under the Apache License 2.0.

Apache Lucene is an information retrieval library written in Java. It provides indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities. It is released under the Apache License 2.0.

6 `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
7 `http://www.apache.org/licenses/LICENSE-2.0.html`
8 `http://www.osgi.org/Specifications/HomePage`
9 `http://www.gnu.org/licenses/gpl.html`

Apache Maven is a build automation tool for to build and manage projects written in JAVA and other languages. Maven serves a similar purpose to the Apache Ant[10] tool, but it is based on different concepts and works in a different manner. It is written in JAVA and released under the Apache License 2.0.

Apache Mina is a JAVA network application framework written in JAVA. Mina provides unified APIs for various transport protocols. A user application interacts with Mina APIs, shielding the user application from low-level I/O details. It is has the Apache License 2.0.

Apache OpenJPA is an implementation of the JAVA Persistence API specification.[11] It is an object-relational mapping solution for the JAVA language, which simplifies storing objects in a database. It is written in JAVA and released under the Apache License 2.0.

The choice of using only OSS systems might limit the generalizability of our findings. Even though we strive to build techniques that do not rely on features specific to open source systems, we cannot claim they will equally work in closed source and industrial settings. However, this threat to validity regards mostly the results of our analyses (such as the defect prediction analyses presented in Chapter 5), and it does not affect the effectiveness of the mining techniques, which can be applied to data from industrial contexts and provide similar performances.

3.5 Summary

In this chapter we presented the high-level methodology that we followed in our dissertation to provide evidence toward our thesis. We also presented how we attempted to create studies that can be replicated, through the dissemination of our benchmarks, and by using OSS systems as subjects of our studies. In the following chapters we describe the different experiments and case studies we did following this approach. In each chapter we follow a methodology of its own to answer the questions we raise or to evaluate the techniques we propose. We allocate the chapters in three other parts: In Part II we address the first challenge in mining unstructured software data (*i.e.*, reconnecting it with code artifacts), in Part III we present techniques for giving a structure to unstructured data so that we can use more appropriate mining techniques afterward, and in Part IV we conclude by taking a step back from our research and by proposing future work.

10 `http://ant.apache.org/`
11 `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`

Part II

Linking Unstructured Data and Source Code Artifacts

The vast majority of mining software repositories approaches start from the source code. Once the information extracted from the source code is available and correctly modeled, it is usually enriched with information extracted from other software repositories, such as those with versioning system data or issue tracking system data. This method has been proved very effective [52].

Manually keeping links between artifacts in software repositories is hard: It constitutes a tedious, error-prone, and time-consuming task to be done gradually. It forces developers to interrupt their normal programming flow. For these reasons, in software repositories we often find no predetermined links between one software repository to another. To tackle this problem, researchers have proposed a number of well-tested linking techniques. Some of them were devoted to automatically recover the traceability links *between unstructured data and other artifacts* (e.g., *[125]).*

In this part of the dissertation we present our work toward reconnecting emails to development artifacts, so that this information can be used for supporting program comprehension and software evolution analysis. We present novel techniques for recovering traceability links and we compare them to the state of the art. Subsequently we show that these traceability links can be used in different software engineering scenarios.

In Chapter 4 we present our first step, in which we first devised lightweight linking techniques to reconnect email and source code, then we compare them to the state of the art in traceability for unstructured data. Once we determined that our techniques are the best performing, we use them in two scenarios. In Chapter 5 we use them to create new metrics to seize a novel aspect of the software process and use them to improve existing defect predictions models. In Chapter 6 we use the links to integrate email data in the development environment, and we show that this novel information, if correctly provided and displayed to developers, supports program comprehension.

Chapter 4

Recovering Traceability Links Between Emails and Source Code Artifacts

In our thesis statement (see Section 1.2), we presented the two main challenges in mining unstructured data, in particular development emails. In this chapter we present an investigation in which we devised and evaluated a number of techniques to tackle the first challenge: the disconnection between email artifacts and code artifacts.

4.1 Overview

Email messages often reference other data sources, such as source code, but there is no actual link to referenced artifacts. Emails can provide new data and metrics to enrich the already available information [200] (*e.g.*, emails pertaining to certain entities of the source code can integrate, or supply, incomplete *documentation*), but they must first be *linked* to the discussed source code artifacts. Connecting emails to the source code can be helpful for various tasks [3]:

- *Understanding software systems:* As systems are continuously growing in complexity and size, they help both bottom-up and top-down comprehension [3];
- *Recovering design rationales:* Often, developers discuss design decisions over mailing lists [23]. Establishing the link between software entities and those discussions permits to join design decisions and their implementation;
- *Performing impact analysis:* After a change is discussed and approved, it is implemented. Tracing these discussions with the subsequent code modifications gives hints about the impact of changes.
- *Identifying coupling*: Code entities that are often mentioned at the same time are implicitly coupled.
- *Extracting developer behavior:* Provided the appropriate links, it is possible to verify how changes occur in the source code (*e.g.*, if they are discussed before or after their implementation).
- *Examining socio-technical congruence:* Provided the connection between code artifacts and email communication, we can investigate the alignment between the technical dependencies and the social coordination in a project.

To gain such benefits, the link between source code and emails must be present, up-to-date, and relevant. In this chapter, we investigate whether and how effectively these links can be *automatically* established. In the first phase, we devise lightweight lexical methods, based on textual matching, to establish the link between emails and source code artifacts; and we evaluate our methods against a manually created benchmark containing correct links between emails and source code of a JAVA software system. In the second phase, we consider the state of the art in recovering traceability links in software engineering (which mainly focuses on recovering links between source code and authoritative documents, *e.g.*, system documentation) and compare its performances against our best performing methods, using another manually created benchmark comprising six OSS systems written in four programming languages.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We identify the importance of reconnecting development emails to source code artifacts*. This section showed the objectives that we achieve by recovering traceability links between emails and code artifacts.
- *We devise, implement, and test a number of lightweight approaches to recover the traceability links between emails and source code artifacts*. The approaches we devise can be used to retrieve the links in real-time, thus can be easily integrated in development environments.
- *We conduct a comprehensive evaluation of the state-of-the-art linking techniques*. We compare existing linking methods ranging from our lightweight lexical approaches to the state of the art in recovering traceability links from authoritative documents, in which researchers effectively adapted approaches from the IR field [3; 126].
- *We produce two benchmarks for evaluating the recovery of traceability links between emails and source code artifacts*. We create the first by analyzing the mailing list of a JAVA software system and linking emails to the last source code release, and we create the second by analyzing the mailing lists of six software systems, written in four different programming languages. For each benchmark, we manually annotated a statistically significant number of emails.

Structure of the chapter. In Section 4.2 we illustrate how we created the necessary benchmark through MILER (see Section 3.2.1), our toolset for exploring email data. In Section 4.3 we detail the lightweight approaches we devised, analyze their theoretical complexity, and evaluate their effectiveness. In Section 4.4, we present the IR methods that shown to be effective for traceability in software engineering, and that we apply in the context of emails. In Section 4.5 we compare our lightweight approaches to the state of the art, discuss the experiment, and list the potential threats to the validity of our experiments. In Section 4.6 we review the related work. We conclude by summarizing our contributions in Section 4.7.

4.2 Benchmark Creation With the Miler Toolset

Our work aims at finding links between two artifacts produced during software development: source code and mailing lists. We want to verify whether it is possible to automatically—and efficiently—find reliable traceability links between emails and software entities using lightweight lexical approaches (namely text and regular expression matching), which exploit intrinsic characteristics of source code elements, rather than expensive IR models or NLP. We also want to compare our methods with more sophisticated IR techniques that led to valuable results in a similar context.

The state of the art mainly spans vector space model (VSM) [123] and latent semantic indexing (LSI) [60] (see Section 4.6); these are currently the methods against which new techniques have to be tested. While researchers successfully adapted these techniques from information retrieval, they mainly applied them to authoritative documents (*e.g.*, functional requirements): We cannot assume that these techniques will provide the same results when applied to email data. In fact, emails have peculiarities that may alter the results (*e.g.*, they contain low level information, such as explicit references to classes, that, for instance, requirements do not include). We argue that a specific *benchmark* is needed to reliably validate and compare approaches against each other in this domain. In the following, we describe the benchmark we created.

We conducted our experiments in two phases: In the first phase, we devised, implemented, and evaluated lightweight lexical approaches for retrieving traceability links between emails and source code entities. In the second phase, we compared our best performing lightweight techniques with the state of the art in traceability.

To reduce overfitting on the data, we created two different benchmarks for the two phases. In the first benchmark (B1), used to test our lightweight methods in the first instance, we considered one Java system (ArgoUML) in its last release and linked its classes to emails taken from its developments and user mailing lists. In the second benchmark (B2), used to compare our best lightweight methods to the state of the art in traceability, we analyzed six unrelated OSS systems, written in four different programming languages, and linked their code entities to emails taken from their development mailing lists only.

To create reliable benchmarks we used Miler (see Section 3.2).

4.2.1 Importing Email Data

Using the MarkMail importer described in Section 3.2.1, we extracted all the emails from the selected mailing lists, and modeled them in the Miler kernel, according to our meta-model. Table 4.1 presents the email datasets considered for each system and benchmark.

Table 4.1: Emails per benchmark and software system

Benchmark	System	Programming Language	Emails			
			Population	Sample		
				Size	Confidence	Error
B1	ArgoUML 0.28	Java	29,024	3,000	95%	1.7%
B2	ArgoUML	Java	29,112	355	95%	5.0%
B2	Freenet	Java	26,412	379	95%	5.0%
B2	JMeter	Java	20,554	380	95%	5.0%
B2	Away3D	ActionScript	9,757	370	95%	5.0%
B2	Habari	PHP	13,095	374	95%	5.0%
B2	Augeas	C	2,219	281	95%	5.0%

Since we have no prior details about the distribution of traceability links in email archives, we employ random sampling without replacement (as opposed to other techniques, *e.g.*, stratified random sampling) to extract reliable sample sets from the populations of the emails. We establish the size (n) of such sets with the following formula [193]:

$$n = \frac{N \cdot \hat{p}\hat{q} \left(z_{\alpha/2}\right)^2}{(N-1)\, E^2 + \hat{p}\hat{q} \left(z_{\alpha/2}\right)^2} \tag{4.1}$$

This formula includes the *finite population correction factor*, because we consider populations that are statistically relatively small. N is the size of the considered population (*e.g.*, 20,554 emails for JMeter); $\hat{p}$ is the *expected* proportion of emails referring a specific source code entity in the sample set, while $\hat{q}$ is $(1 - \hat{p})$ (*i.e.*, emails *not* referring that entity); E is the *margin of error* to be considered, and $z_{\alpha/2}$ is the *critical value* [193] associated to the chosen *confidence level*.

In our experiment, since the proportion ($\hat{p}$) of the emails referring to a specific entity of the source code is not known *a priori*, we consider the worst case scenario (*i.e.*, $\hat{p} \cdot \hat{q} = 0.25$). For the first benchmark, which is based only on one system, we keep a confidence level of 95% and an error (E) of 1.7%; while for the second benchmark, which considers more systems, we keep a confidence level of 95% and an error (E) of 5%. In practice, this means that if a source code entity is cited in f% of the sample set emails, we are 95% confident it is cited in the $f\% \pm E$ of the population emails. This validates the quality of this sample set as an exemplification of the entire population; it is not directly related to the *precision* and *recall* values presented later, which are actual values based on manually analyzed elements. Applying this formula to our populations results in the sample sizes (n) shown in Table 4.1.

4.2.2 Importing Source Code Data

The other ingredient of our benchmarks is the source code. In the first benchmark (B1), we consider only the release 0.28 (March 2009) of ArgoUML, while for the second benchmark (B2), we take all system versions throughout the systems' history. Using our *revision importer* (see Section 3.2), we take the chosen releases (*e.g.*, 0.28 for ArgoUML in B1, or all the official releases for ArgoUML or JMeter in B2), otherwise we use the *checkout by date* feature of the version control system (*i.e.*, we retrieved the committed code in intervals of 3 months, starting 3 months after repository creation). As opposed to Antoniol *et al.* [3] and Marcus *et al.* [126], we do not consider files as the unit for documents, but we link emails with source code entities: *classes* for object-oriented systems, *functions* and *structures* for procedural language systems. To achieve this, we parse the source code, extract the model, and find the links between model entities and emails. Table 4.2 lists the collected data.

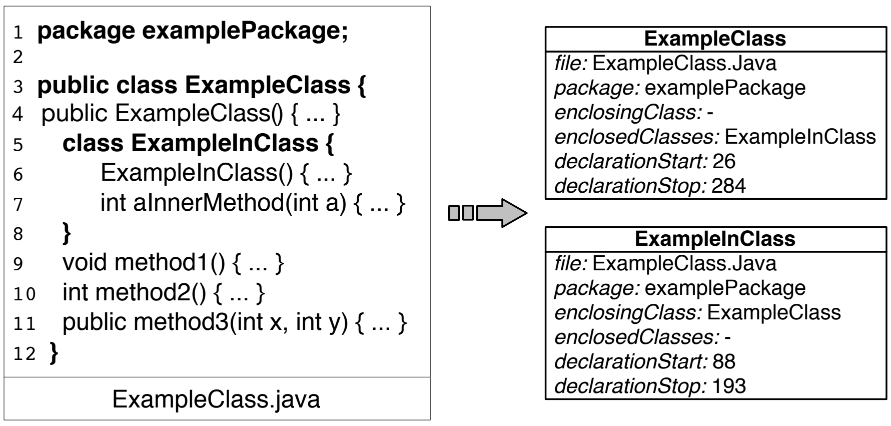
To reduce the threats to external validity of our experiments, *i.e.*, to improve the generalizability of our findings, in benchmark B2 we consider systems written in different languages. To locate and model the source code from files, we use the code importers described in Section 3.2. In particular, we use the industrial tool inFusion, for Java and C systems. For the other languages we implemented our own code fact extractors.

Since we do not need all the information that a full-fledged parser is able to provide (*e.g.*, inFusion also performs static analysis of the code—not necessary in this context), we limit ourselves

Table 4.2: Source code entities per benchmark and software system

Benchmark	System	Number of Releases	Number of Entities		
			First release	Last Release	Total
B1	ArgoUML 0.28	1	n/a	2,197	2,197
B2	ArgoUML	11	906	2,396	18,252
B2	Freenet	30	822	2,026	37,878
B2	JMeter	20	16	906	11,105
B2	Away3D	9	132	465	2,351
B2	Habari	12	20	124	1,105
B2	Augeas	17	60	675	8,042

to the necessary features. This has the advantage of being more lightweight both from an implementation point of view and in terms of parsing efficiency. We do this by implementing dedicated island parsers [136] in our PetitIsland framework (see Section 8.7).



```
package examplePackage;

public class ExampleClass {
 public ExampleClass() { ... }
   class ExampleInClass {
        ExampleInClass() { ... }
        int aInnerMethod(int a) { ... }
   }
   void method1() { ... }
   int method2() { ... }
   public method3(int x, int y) { ... }
}
```

ExampleClass.java

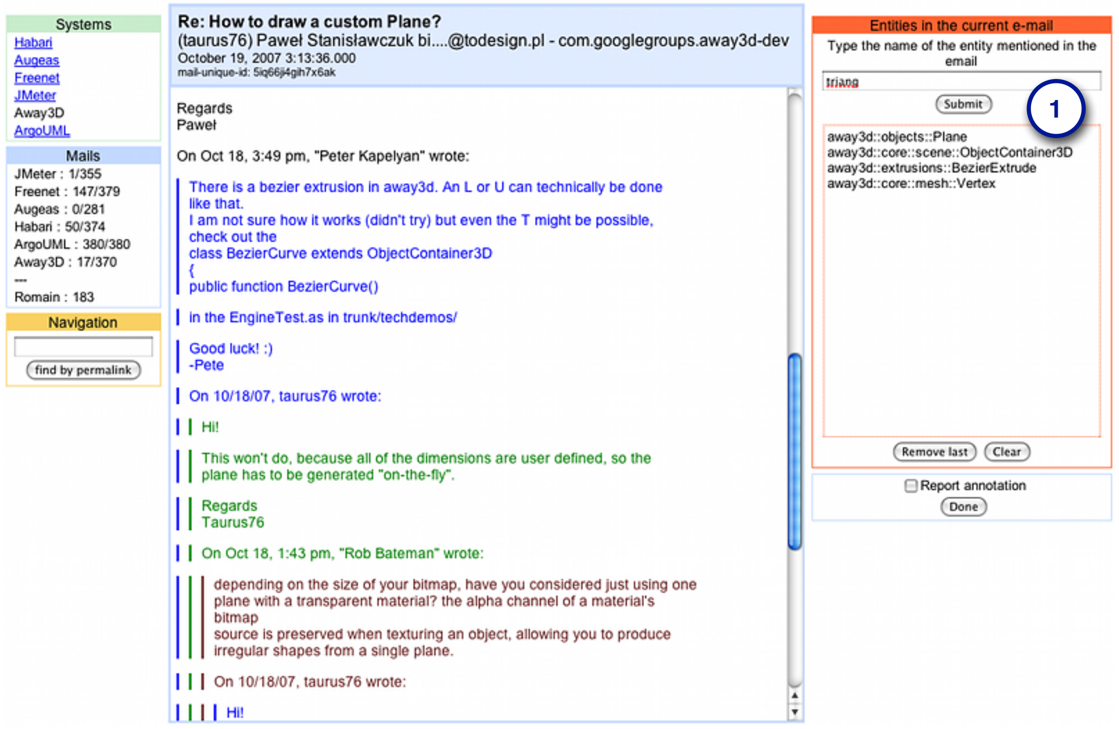
ExampleClass
file: ExampleClass.Java
package: examplePackage
enclosingClass: -
enclosedClasses: ExampleInClass
declarationStart: 26
declarationStop: 284

ExampleInClass
file: ExampleClass.Java
package: examplePackage
enclosingClass: ExampleClass
enclosedClasses: -
declarationStart: 88
declarationStop: 193

Figure 4.1: Example of parsed information and the resulting model

From the source code of an entity, we require the following information: in the case of our lightweight lexical methods (see Section 4.3), we only need to extract the name, the containing package, and the location of the entities to be linked; for IR techniques (see Section 4.4), we additionally need the terms included in entity declarations, to generate the term vectors.

Figure 4.1 shows an example of what information is parsed and the resulting model. Bold parts are the relevant facts (*i.e.*, island code); they constitute the information in the model. The other parts (*i.e.*, water) are not parsed, but a reference is stored in the model, so that contained terms can be later retrieved. On the right hand side of the figure, for instance, we know where `ExampleInClass` is defined—in which file, where it declaration begins and ends—thus we can extract only the terms involved in its definition, excluding terms defined in the rest of the document.



Systems
Habari
Augeas
Freenet
JMeter
Away3D
ArgoUML

Mails
JMeter : 1/355
Freenet : 147/379
Augeas : 0/281
Habari : 50/374
ArgoUML : 380/380
Away3D : 17/370

Romain : 183

Navigation
find by permalink

Re: How to draw a custom Plane?
(taurus76) Paweł Stanisławczuk bi....@todesign.pl - com.googlegroups.away3d-dev
October 19, 2007 3:13:36.000
mail-unique-id: 5iq66jj4gih7x6ak

Regards
Paweł

On Oct 18, 3:49 pm, "Peter Kapelyan" wrote:

> There is a bezier extrusion in away3d. An L or U can technically be done
> like that.
> I am not sure how it works (didn't try) but even the T might be possible,
> check out the
> class BezierCurve extends ObjectContainer3D
> {
> public function BezierCurve()
>
> in the EngineTest.as in trunk/techdemos/
>
> Good luck! :)
> -Pete
>
> On 10/18/07, taurus76 wrote:
>
> > Hi!
> >
> > This won't do, because all of the dimensions are user defined, so the
> > plane has to be generated "on-the-fly".
> >
> > Regards
> > Taurus76
> >
> > On Oct 18, 1:43 pm, "Rob Bateman" wrote:
> >
> > > depending on the size of your bitmap, have you considered just using one
> > > plane with a transparent material? the alpha channel of a material's
> > > bitmap
> > > source is preserved when texturing an object, allowing you to produce
> > > irregular shapes from a single plane.
> > >
> > > On 10/18/07, taurus76 wrote:
> > >
> > > > Hi!

Entities in the current e-mail
Type the name of the entity mentioned in the email
triang
Submit

away3d::objects::Plane
away3d::core::scene::ObjectContainer3D
away3d::extrusions::BezierExtrude
away3d::core::mesh::Vertex

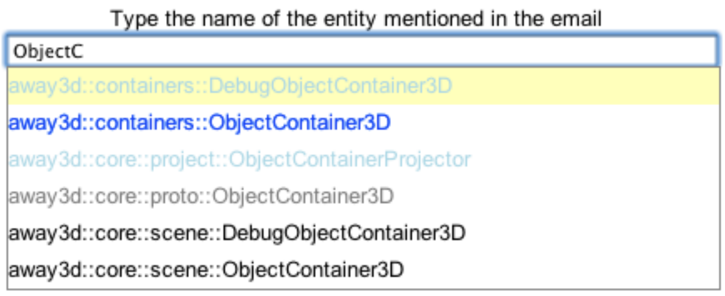
Remove last | Clear

☐ Report annotation
Done

Figure 4.2: Miler Game: The Web Application for Creating the Benchmark

4.2.3 Manual Benchmark Creation With the Miler Game

Creating the benchmark requires one to read all the emails in the sample sets and to annotate them with code entities discussed therein. We extended the Miler Game (see Section 3.3) to assist the manual linking task. Figure 4.2 shows the Miler Game's main page, after a user logged in. With respect to the main interface of the Miler Game, we added a new panel, called *Annotation panel* (Point 1), which contains the list of the code entities linked up to the present.



Type the name of the entity mentioned in the email
ObjectC
away3d::containers::DebugObjectContainer3D
away3d::containers::ObjectContainer3D
away3d::core::project::ObjectContainerProjector
away3d::core::proto::ObjectContainer3D
away3d::core::scene::DebugObjectContainer3D
away3d::core::scene::ObjectContainer3D

Figure 4.3: Miler Game: The *autocompletion* field

The *Annotation panel* features an autocompletion field (Figure 4.3) to help users to link an email to the correct code entities. Users can see any entity whose name includes the typed letters; the autocompletion avoids typos since only existing entities can be linked. The field displays entity names with the following coloring convention: Entities are black if present in the last release before the email date (this is the only option for B1, which only uses one release of ArgoUML); light-gray if present only in older releases; blue if implemented in the first release after the email date; and light blue if released later. In Figure 4.2, the user typed "ObjectContainer3D". The menu shows the homonymous entities in three colors: [...]`proto::ObjectContainer3D` is light-gray, as it is not in the current release, but only in older ones; [..]`containers::ObjectContainer3D` is blue, because it will be created only in the future release; [..]`scene::ObjectContainer3D` is black, because it exists in the current release. This helps choosing the most appropriate entity.

Six members of the REVEAL research group, with several years of programming experience, inspected the sample sets. In both benchmarks, the emails were randomly divided in overlapping sets, resulting in 49% of the messages analyzed by two people. A complete agreement was reached on 91% of these messages, with the remaining annotations featuring small differences: Almost all the divergences were caused by one of the two reviewers missing to annotate a link that was actually present in the email. All the errors were corrected.

Annotators did not differentiate between links only present in text quoted from previous messages and present in the new content of the email. This allows the usage of these benchmarks as a general case of textual information containing source code identifiers and discussions.

4.2.4 Evaluation

To compare the effectiveness of the approaches, we measure two well known IR metrics for the quality of the results [123] (*i.e., precision* and *recall*), based on the following definitions:

- **True Positives** (*TP*): elements that are correctly retrieved by the approach under analysis (*i.e.,* links to source entities also present in the oracle)
- **False Positives** (*FP*): elements that are wrongly retrieved by the approach under analysis (*i.e.,* links to source entities not present in the oracle)
- **False Negatives** (*FN*): elements that are not retrieved by the approach under analysis (*i.e.,* links to source entities only present in the oracle)

Standard formulas for calculating precision and recall are:

$$Precision = \frac{|TP|}{|TP + FP|} \tag{4.2}$$

$$Recall = \frac{|TP|}{|TP + FN|} \tag{4.3}$$

The union of TP and FN constitutes the set of correct links present in the benchmark per email, while the union of TP and FP constitutes the set of links retrieved by the used approach. In short, *precision* is the fraction of the retrieved links that are correct, while *recall* is the fraction of the correct links retrieved.

A number of emails in the benchmark have no references to source code entities, thus the union of TP and FN is empty. In these cases, the denominator in the recall formula is zero and the *recall* value cannot be calculated. Analogously, it is possible for automatic approaches not to find any link between an email and source code. In this case, the *precision* value cannot be evaluated because the denominator in the corresponding formula is equal to zero. To overcome these issues, we first calculate the average of TP, FP, and FN, on the entire dataset. Then, we measure the average *precision* and *recall* from those values. This solution also takes into account the impact of false positives on *precision*, when the set of benchmark references is empty. Antoniol *et al.*, who encountered the same difficulty, used a similar approach [3].

Precision (P) and recall (R) are two quantities that trade off against one another: Intuitively, it is possible to link each mail with all classes, reaching a recall value of 1, but a very low precision. For this reason, in order to measure such trade-off we added the *F measure*, which is the weighted harmonic mean of precision and recall:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}, \beta^2 = \frac{1-\alpha}{\alpha} \longrightarrow F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{4.4}$$

The weighting of precision and recall can be decided through the value of β. We decided to emphasize neither the recall nor the precision, because our approaches can be used in many different situations, and it is up to the engineer to select the most appropriate one. Thus, we prefer to give a general view of the result: We use a β value of 1 to obtain the *balanced F measure*.

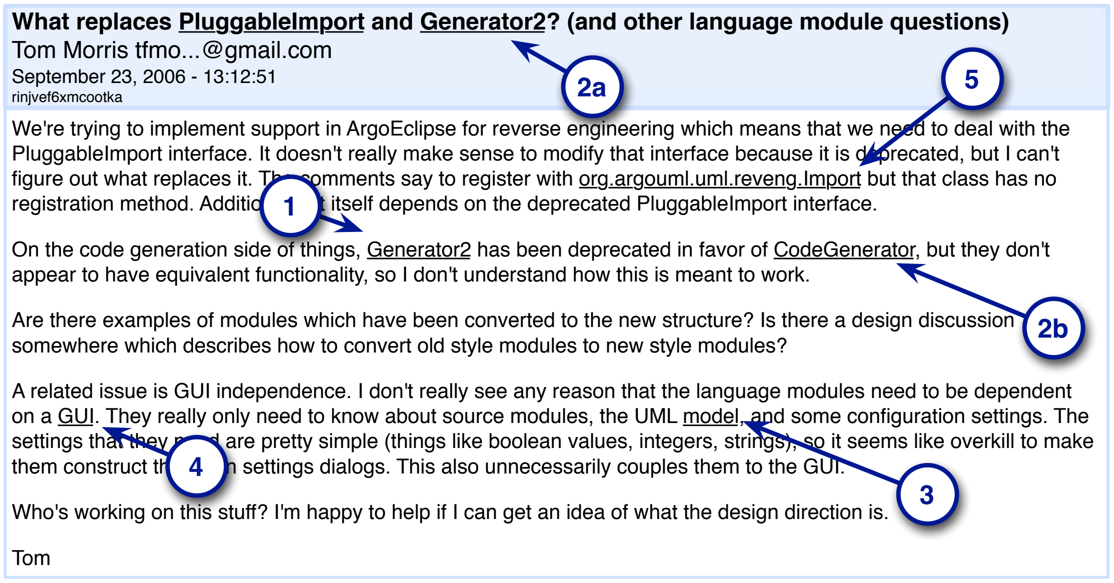
4.3 Lightweight Traceability Linking

In this section we analyze the different lightweight approaches we devised and tested using our first benchmark B1 (see Section 4.2). We illustrate each approach explaining the rationale, showing the implementation, presenting the results achieved, and discussing the computational complexity of the most elaborate ones. All techniques go from artifact to emails: We start from a given artifact, extract the necessary information from the MILER kernel (*e.g.*, artifact name), and use this information to establish whether each email contains a link to said artifact. To illustrate our techniques, we use an email taken from the sample set (Figure 4.4), which contains several examples of how people reference classes in emails.

Why efficiency matters. We implemented our methods to be (1) *compact*, (2) *simple*, and (3) *fast*. We are interested in these features, because we strive for *practicality*, *i.e.*, we want our techniques to be usable in a real-world setting. As a typical usage scenario of traceability links between source code and emails, we expect not only software engineering researchers, but also developers who search for emails that discuss a code artifact they are currently maintaining.

In such a scenario, developers work on a non-specialized multi-purpose computer. The linking methods must be: (1) *compact*, thus not occupying large amount of disk space to store information; (2) *simple*, thus being easy to implement and include in different working environments; and (3) *fast*, presenting results in a few seconds at most, thus not requiring any large computational effort.

In the following we not only present the rationale behind our techniques, but we also study their complexity and present efficiency improvements, whenever possible.



What replaces PluggableImport and Generator2? (and other language module questions)
Tom Morris tfmo...@gmail.com
September 23, 2006 - 13:12:51
rinjvef6xmcootka

We're trying to implement support in ArgoEclipse for reverse engineering which means that we need to deal with the PluggableImport interface. It doesn't really make sense to modify that interface because it is deprecated, but I can't figure out what replaces it. Th comments say to register with org.argouml.uml.reveng.Import but that class has no registration method. Additio t itself depends on the deprecated PluggableImport interface.

On the code generation side of things, Generator2 has been deprecated in favor of CodeGenerator, but they don't appear to have equivalent functionality, so I don't understand how this is meant to work.

Are there examples of modules which have been converted to the new structure? Is there a design discussion somewhere which describes how to convert old style modules to new style modules?

A related issue is GUI independence. I don't really see any reason that the language modules need to be dependent on a GUI. They really only need to know about source modules, the UML model, and some configuration settings. The settings that they d are pretty simple (things like boolean values, integers, strings), so it seems like overkill to make them construct th n settings dialogs. This also unnecessarily couples them to the GUI.

Who's working on this stuff? I'm happy to help if I can get an idea of what the design direction is.

Tom

Figure 4.4: An email from the sample set containing various kinds of references to entities

4.3.1 Artifact Name, Case Insensitive

Intuition: The simplest way for an email author to reference an entity is using its name. As an example, in Figure 4.4 Point 1, the class `Generator2` is simply mentioned by name. Given an artifact, this method links all the emails that contain at least one string corresponding to its name. This method also considers valid the strings that do not respect the original case of the letters in the artifact name; in fact, when quickly writing about an artifact, email authors might not respect naming conventions (*e.g.*, upper case letters). Finally, this method does not impose any restriction on the characters surrounding the candidate string. As we see in Figure 4.4 Points 2a and 2b, where class names `CodeGenerator` and `Generator2` are followed by punctuation, authors can include names in different contexts.

Implementation: This simple implementation consists in verifying, given an artifact name, whether there is at least one string corresponding to it in the considered email. When traversing the email content, the implementation takes into consideration neither the case of the artifact name nor the preceding or following characters. Since Miler kernel provides the name of an artifact, this implementation can be reduced to a string-matching problem [50].

Precision: 0.09 – Recall: 0.70 – F-Measure: 0.16

Results: The most interesting result of this simple match is the recall value: It reaches a value of 0.70 on our statistically representative sample. The trade-off is a low precision, due to the many false positives.

Considering Figure 4.4 Point 3, we note that the word "*model*" does not refer to the class `Model` in the package `org.argouml.model`, as this simple approach wrongly assumes. This is one of the many examples that make this approach generating false positives.

All the matching techniques that follow use this simple class name match as the first step for their implementation: They all consider the class name and require it to be present in one single word in the email content. For this reason, the value reached with this first approach is the upper bound of the recall.

Complexity: O(a+m), where a stands for the number of characters in the artifact name and m the number of characters in the emails. We repeatedly check the same artifact name against many emails, thus the string matching algorithm by Knuth *et al.* [108] is an optimal solution.

4.3.2 Artifact Name, Case Sensitive

Intuition: In many object-oriented languages, developers follow the widely accepted convention of starting class names with a capital letter and use *media capitals* (*e.g.*, `ClassName`), also known as *CamelCasing*, to compound words. For this reason, case sensitivity might be important when matching a class name against an email text. For example, in Figure 4.4 Point 3, we note that the author of the email is *not* referring to the class `org.argouml.model.Model`, but is using the term "*Model*" with its common dictionary meaning. By considering case sensitivity, this email would not be reported as relevant when searching for email discussing the class `Model`, while it would be recommended for the classes `CodeGenerator` and `Generator2`.

Implementation: In this approach we use an algorithm similar to the previous one, but we do consider the case of characters. We expect to reduce the false positives that the previous algorithm generates.

Precision: 0.33 – Recall: 0.69 – F-Measure: 0.46

Results: As expected, the recall value did not decrease significantly. On the contrary, the simple additional case sensitivity check greatly increases precision (*i.e.*, by 24%). The number of false positives dropped, while the number of good links not retrieved was almost as high as in the previous approach.

This result points out that class names are mainly mentioned respecting camel casing. This simple check thus helps to separate common words of discussions from true references to source code entities.

One of the false positive created by this approach is the one marked with point 4 in Figure 4.4. The word `GUI` is not a reference to the class `org.argouml.ui.GUI`, on the contrary the author is writing about a component of the ArgoUML application, from a user point of view. Also, the class `Generator` is wrongly recognized as being referenced, because its name is part of the word "*Generator2*" or "*CodeGenerator*" (Figure 4.4 points 1 and 2, respectively).

Complexity: O(a+m), as with the previous technique.

4.3.3 Strict Regular Expression

Intuition: This method exploits some peculiar characteristics of source artifacts (in addition to the name) to reach a high precision value. It is an indicator of the upper bound for precision. First, this method verifies whether the chosen email contains a string corresponding to the artifact name, if so, it analyzes the surrounding text. We consider the text *after* the string corresponding to the artifact name:

- **File extension:** Many programming languages store source code and binary files with recognizable extensions, often named after the code artifact they contain. For example, JAVA classes are stored in files with special extensions (*i.e.*, `java` and `class`) and the compiler enforces these files to have the same name as the class they define. For this, given the artifact, when we find a string corresponding to its name and followed by these extensions, we can be almost certain that it is a valid traceability link.
- **Whitespace and punctuation:** When email authors discuss about an entity, and not about its containing file, the file extension is likely to be omitted. When there is no extension, we impose that, after the string corresponding to a class name, there must be an empty space or a punctuation sign. This constraint both admits natural language text punctuations and resolves ambiguities in strings with the same prefix (*e.g.*, when we search for emails related to the class `Model`, the previous approach would also link any email containing the string `ModelFacade`, while this method would not).

We analyze the text to be found *before* strings that correspond to artifact names:

- **Package:** In most object-oriented languages, every class is in a package; two classes can have the same name only if they reside in different packages. We require the last part of the package name to be present before the string corresponding to a class name. Such a strict requirement resolves the ambiguity of different classes having the same name and guarantees high precision.
- **Punctuation:** Package identifiers are commonly separated by a dot (*e.g.*, `org.argouml`). In many languages, packages are represented in the file system as directories (*e.g.*, `org/`, `org/argouml/`). For this reason, email authors referring to a class in its file form could divide the package identifiers through the common file system separators, *i.e.*, "\" and "/". We require the last part of the package to be preceded with this special punctuation.
- **Whitespace:** We require the text that precedes the aforementioned punctuation to be either a whitespace (*e.g.*, blanks, tabulator keys, new lines) or the rest of the package preceded by whitespaces.

By enforcing these constraints, the example email will be reported as a valid link only when searching for emails related to the class [...]`reveng.Import` (being correctly referred in Figure 4.4, Point 5). No false positive link is produced.

Implementation: The core of this approach consists in a regular expression, which we express here according to the IEEE POSIX Basic Regular Expressions (BRE) standard:

```
(.*)(\s)
(<beginning of package>(.|\\|/))?
<last part of package>(.|\\|/)
<entity name>
(\.(java|class|cs|h|c|php|as)|[:punct:]|\s)
(.*)
```

MILER replaces the parts in italic and angle brackets (*e.g.*, `<class name>`) with the code artifact's specific information available in the Miler kernel, and generates the complete regular expression text. For example, when applied to the JAVA class `org.argouml.application.Main`, this function produces the following regular expression:

```
(.*)(\s)
(org(.|\\|/)argouml(.|\\|/))?
application(.|\\|/)
Main(\.(java|class)|[:punct:]|\s)
(.*)
```

Precision: 0.94 – Recall: 0.10 – F-Measure: 0.18

Results: Using this strict match, the recall value radically changes and reaches a very low value. On the other hand, as we expected, the precision reaches a top value. Due to the small number of links that this approach retrieves, a very few false positives (*i.e.*, 11 for the whole data set) are sufficient to lower the precision from 1 to 0.94. Classes that are not in the ArgoUML model, but have the same name and last part of package, are recognized as references with this approach, thus causing those false positives. The F-Measure value is as low as in the match based on the simple class name approach, not case sensitive. There is no difference in the results when using case sensitivity.

Complexity: O(a $\cdot$ m), where a sums the characters in the artifact name and in the package, and those used in the regular expression; m sums the email characters. Since retrieving the information from Miler kernel is done in constant time, the auxiliary function that generates the regular expression text takes $\Theta(a)$ time.

In our experiment, we generate the regular expression text only once per entity and we use it to match multiple emails. The regular expression always contains the complete class name in its text. Checking for the presence of the artifact name is significantly faster than evaluating the complete regular expression, thus, we first check for the presence of the artifact name: if it appears, we invoke the regular expression matcher.

The regular expression matcher dominates the computational time of this approach. The fastest algorithm to match a text against a regular expression is based on work by McNaughton and Yamada [131] and by Thompson [190]. They demonstrated that any regular expression could be represented with a nondeterministic finite automaton (NFA), which can be matched against a text in polynomial time. In the worst case, the regular expression matching is computed on all the m characters of the email requiring $O(m \cdot a)$ time.

Further efficiency improvement: Rabin and Scott showed that any NFA can be simulated by deterministic finite automatons (DFA) in which each DFA state corresponds to a set of NFA states [130]. A DFA has the advantage to be matched against m characters in linear time $O(m)$. The most expensive part of this approach is transforming an NFA into the corresponding (potentially much larger) DFA. In the typical case (*i.e.*, our case), the transformation requires $O(a^3)$ time.

When m exceeds a^3, we benefit from using the DFA. For example, in the case of our benchmark, which consists of 2,000 emails, the transformation from NFA to DFA brings an advantage. If we consider the previously cited regular expression, composed of 98 characters, we obtain a^3 = 941,192. Even with only 471 (*i.e.*, 941,192 characters divided by 2,000 emails) characters per email on average, we would have enough text to justify the transformation.

In other words, when using this approach on a large set of emails, we can optimize it to be executed in $O(m + a^3)$.

4.3.4 Loose Regular Expression, Case Sensitive

Intuition: The strictest requirement of the previous approach is the presence, in the matching string, of the package tail before the artifact name. By relaxing this constraint we expect to achieve a better compromise between precision and recall. At the place of the package tail, we allow either an empty space or the complete package string. For the text after the artifact name, we apply the constraints of the previous approach.

This approach would report the example email in Figure 4.4 as valid, for the classes mentioned in Points 1, 2 and 5.

Implementation: As in the previous approach, we implement this technique with a regular expression.

```
(.*)
((\s)|(<package>(.|\\|/)))
<class name>
(\.(java|class|cs|h|c|php|as)|[:punct:]|\s)
(.*)
```

By applying this function to the class of the previous example, we obtain the following regular expression:

```
(.*)
((\s)|
(org(.|\\|/)argouml(.|\\|/)application(.|\\|/)))
Main(\.(java|class)|[:punct:]|\s)(.*)
```

Precision: 0.45 – Recall: 0.54 – F-Measure: 0.49

Results: Results show that the recall increased to 42%, while precision lost 49%. The F-Measure, which is slightly higher than the match on the class name case-sensitive, points out that this method is the best choice, up to now, if recall and precision are considered equally important.

One of the false positives, considering Figure 4.4, is the word marked by Point 4. The string matches the regular expression, however the author is not discussing a class named `GUI`.

Complexity: O(a · m), same as above. It can be equally optimized for large sets of emails.

4.3.5 Mixed approaches

These approaches integrate the previous ones to combine their strengths and offer a more convenient trade-off between precision and recall for general cases. They are based on the same rationale as the previous ones: When developers communicate about a code artifact, they often mention it by its name. For this, the first step still consists in searching for a string corresponding to the name of the artifact to be linked.

The main precision problem of searching for strings that correspond to artifact names is *polysemy*, *i.e.*, the capacity for a word to have multiple meanings. A convention to support program comprehension is naming artifacts after the real-world objects they model (*e.g.*, ArgoUML's class

`Text` models the behavior of text); this implies that, in any document, a string corresponding to an artifact name might not actually refer to that artifact but simply have the usual definition we find in a dictionary. The intuition behind the mixed approaches is to use a strict matching technique (*e.g.*, the regular expression based method detailed in Section 4.3.3) when a string can have more meanings than the artifact name (*e.g.*, the word "Text"), and to use a loose technique (*e.g.*, the matching on artifact name method) otherwise.

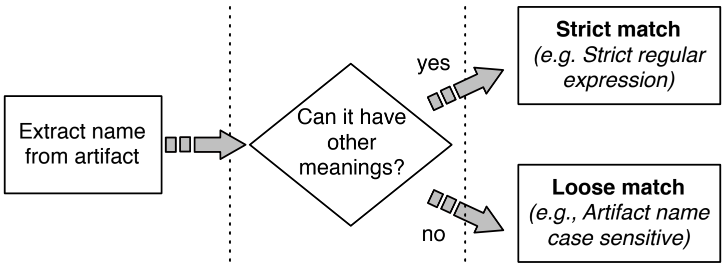

Extract name from artifact
Can it have other meanings?
yes
no
Strict match
(e.g. Strict regular expression)
Loose match
(e.g., Artifact name case sensitive)

Figure 4.5: Mixing approaches to face polysemy-related issues

Figure 4.5 shows our strategy to mix the approaches. First, given an artifact in the Miler kernel, we consider its name: If it might present polysemy, we use a strict matching, otherwise we use a loose matching. The main difference among the following mixed approaches is how they determine whether the artifact name presents polysemy.

Dictionary approach

Intuition: We can find polysemy in those artifact names that also have a meaning in a common dictionary (such as *Dialog*, *Text*, or *Critic*). This approach determines polysemy when the artifact name also appears as an entry in an English dictionary.

Implementation: Given the name of an artifact, a function checks its presence in a dictionary of common English words with more than two hundred thousand words taken from the standard Unix `words` file.[1] If the name is present, the function then uses the strictest matching available (*i.e.*, Strict regular expression, cf. Section 4.3.3) to match the string in the body of an email, otherwise it uses the matching on the Artifact name, case sensitive (cf. Section 4.3.2).

Precision: 0.57 – Recall: 0.62 – F-Measure: 0.60

Results: To obtain a stronger comprehension of the results obtained, we also tried to use the simple matching not case sensitive on the class name (see Section 4.3.1), in place of the case sensitive one. The results are: precision 0.20, recall 0.64. The difference in precision, related to the usage of case sensitivity, is evidence of the fact that the most cited classes are not part of the dictionary. In fact, we recall that the simple match on the class name is only used when the name is not on the dictionary, otherwise we apply the strict regular expression, which is not influenced by case sensitivity.

The F-measure shows that this approach, which makes use of a dictionary to select the severity of the match, is the most effective until now.

1 `http://en.wikipedia.org/wiki/Words_(Unix)`

Complexity: O(a·m). The dictionary can be stored in a hash table, thus string searching is completed in constant time on the number of stored elements. The complexity is dominated by the slowest matching used: in this case the Strict regular expression (see Section 4.3.3).

CamelCase approach

Intuition: Even though dictionaries can be stored in hash tables and searched in constant time, in practice their usage is uncomfortable (*e.g.*, one dictionary per language) and the searching time, although constant, is not zero. This approach exploits naming conventions to determine whether an artifact name can present polysemy.

As previously mentioned, artifacts usually represent abstractions of real-world categories of objects; for this reason, it is common practice to give them names from common words. However, since empty spaces are not allowed in artifact names, whenever an artifact represents a concept which is clearer if named with two or more words, those are compounded, by using media capitals as suggested by naming conventions.

The intuition is that artifact names, which are formed by compounded words, cannot be part of a common dictionary. For example, the artifact name `ExplorerTree` is formed by the two dictionary words `explorer` and `tree`, but their composition is not a dictionary word. For this reason, we suppose that it is sufficient to check for the presence of media capitals to determine whether a word presents polysemy. If the artifact name is a single word, this method uses the loose matching, otherwise, when a name is made by compounded words, the method uses the strict matching.

Implementation: The implementation is analog to the previous algorithm, but in practice, since it does not require a dictionary, it is faster and requires less memory. The test for media capitals considers a word to be compounded if it contains more than one capital letter and at least one lower case letter. In addition to the common cases, this test correctly handles also cases such as the string `PGMLParser`, marked as compounded, and `CSV`, marked as simple.

Precision: 0.63 – Recall: 0.62 – F-Measure: 0.62

Results: The main goal of this approach is to lighten the previous one removing the dictionary search. The results show that, in addition to performance, the precision value increased, without any loss in the recall value.

These results can be explained through a few examples. First, there are class names that are not in a simple English dictionary, but are common computer science terms: `Parser` is one of such terms. The previous approach did not find the string `Parser` in the dictionary, thus it wrongly treated it with the most optimistic match. On the contrary, this new approach found only one capital character and correctly switched to use the strictest match.

Second, there are class names that are part of dictionary words. For example, the class `Init` has a name that is part of many dictionary words (*e.g.*, `Initialization`, `Initial`), thus the most optimistic methods (which does not impose any restriction on characters surrounding the class name) finds wrong matches. On the contrary, this new method correctly treats such words, which have only one capital letter, using the strictest approach.

Finally, classes often have names that correspond to high-level design concepts. For example, `Modeller` is a class name, and a concept in the ArgoUML jargon. It is possible to find it mentioned both as a high level design concept (not referring the class with the same name) and as a class of the system. Also in this case, this new approach takes the right decision of treating this case with the strictest matching. In terms of F-Measure, this method is the most effective.

Complexity: O(a·m). Even though this method has the same linear complexity as the previous one, which uses the dictionary, it consumes less memory and its verification requires a shorter constant time. The Strict regular expression still dominates and determines the overall computational time of $O(a \cdot m)$. This can be optimized up to $O(t + m^3)$, as in previous approaches.

4.3.6 Discussion

Results. In Figure 4.6 we have summarized the results of all approaches. The case insensitive class name approach (A) represents the upper bound for the recall (*i.e.*, 0.70), as all the other approaches we used, as the first step, check for the presence of the class name. Then, they refine the precision using different techniques, which necessarily reduces the recall. It is still possible to implement lightweight techniques to raise it. For example, during the benchmark creation, we noted that, sometimes, class names that are made of compounded words are mentioned in e-mails using those words separately. This aspect can be implemented refining the regular expressions. Heavyweight approaches may find implicit references, thus increasing the recall.

On the opposite extreme, there is the case insensitive strict regular expression approach (C), which reaches top precision but retrieves few links. From the results of the two mixed approaches (E and F), the strict approach was useful to raise the precision value when used together with the case sensitive class name approach.

The two mixed approaches give the best results: They are not only capable of retrieving 62% of all the correct links between e-mails and source code entities, but they also completed this task with the same significant level of precision (*i.e.*, 63% of the links retrieved were true positives).

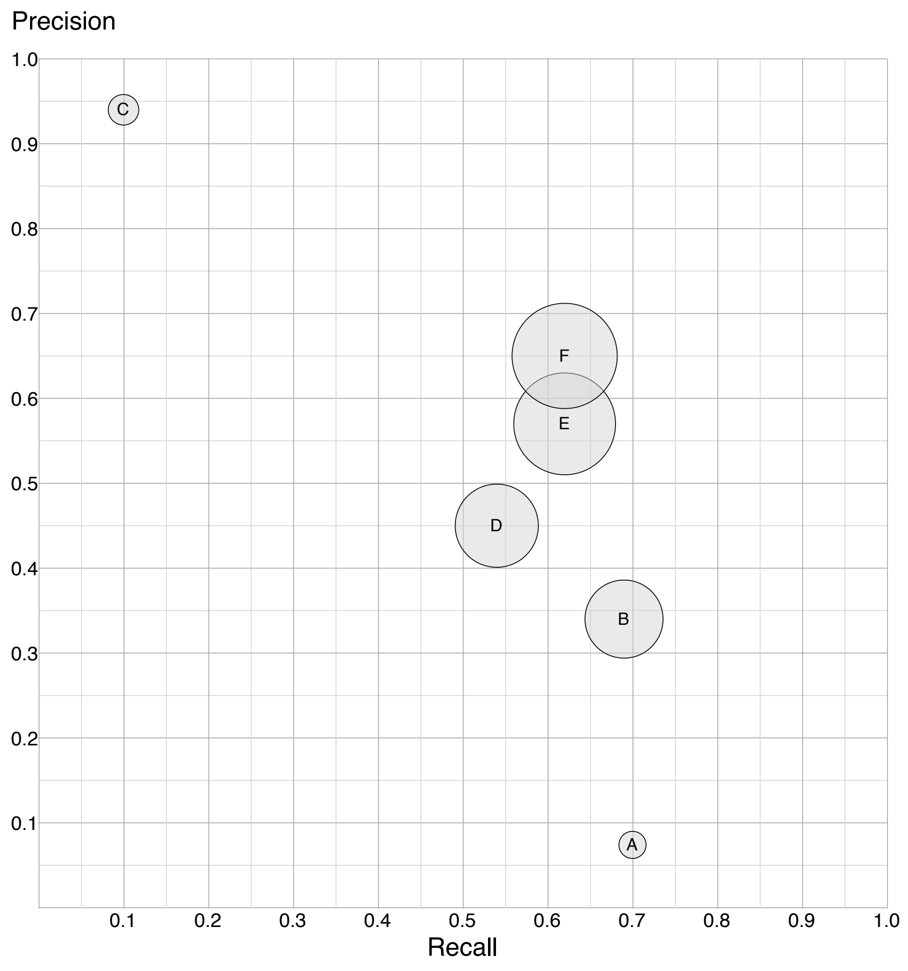
Complexity. All approaches can be computed in polynomial time. In particular, the first two approaches, without regular expressions, are computed in linear time $O(a + m)$, where m is the number of characters in the email, and a the number of characters in the pattern to be searched for (*e.g.*, the artifact name).

The approaches involving regular expressions are computed in time $O(a \cdot m)$, which is still polynomial and, in most of the cases, less than quadratic (it is quadratic when a approaches t, but this happens only for very short emails, that can be checked extremely fast anyways). These results cement the validity of our approaches as lightweight, since they can be both easily implemented and quickly executed.

We showed that these complexities can be even lowered when we plan to use the function on multiple emails. In the approaches involving regular expressions, we illustrated how one can split computation into a pre-processing part and a matching part. The pre-processing is computationally intensive: $O(a^3)$ time in the average case, but the match can be executed in $O(t)$.

4.4 Information Retrieval Techniques

The state of the art in recovering traceability links between code entities and authoritative documents is represented by VSM, introduced in this context by Antoniol *et al.* [3], and by LSI,



	Approach	Precision	Recall	F-measure
A	Artifact name, case insensitive	0.09	0.70	0.16
B	Artifact name, case sensitive	0.33	0.69	0.46
C	Strict regular expression, case insensitive	0.94	0.10	0.18
D	Loose regular expression, case sensitive	0.45	0.54	0.49
E	Mixed, with dictionary, case sensitive	0.57	0.62	0.60
F	Mixed, with CamelCase, case sensitive	0.63	0.62	0.62

Figure 4.6: Precision, recall, and F-measure of all lightweight approaches.

used by Marcus and Maletic [126]. In this section we present these two approaches and their application to the context of retrieving links between code entities and emails.

4.4.1 Vector Space Model (VSM)

VSM approaches represent each document in a corpus as a *term vector*. The size of the vectors is the total number of terms in the corpus' vocabulary and the values reflect the number of occurrences of each term in the document. If we consider a document (d), the cardinality of the vocabulary ($|C|$), and the number of times each term (t_i) occurs in the document, we can define the vector as:

$$v_d = [t_{1(d)}, t_{2(d)}, \dots, t_{C(d)}] \tag{4.5}$$

Methods that use the VSM aggregate and transpose term vectors to compose the *Term Document Matrix* (TDM):

$$\text{TDM} = \begin{array}{c|cccc} & D_1 & D_2 & \cdots & D_N \\ \hline t_1 & 0 & 1 & \cdots & 0 \\ t_2 & 0 & 0 & \cdots & 4 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ t_C & 1 & 2 & \cdots & 0 \end{array}$$

In the example matrix, the term t_C appears once in the first document (D_1) twice in the second, but not in the last (D_N).

Term Weighting. In the TDM, each term is assigned with a weight, *i.e.*, the number of occurrences. To compare documents of different lengths, this weight is normalized on the length of the document. This weighting is called *term frequency* (tf):

$$tf_{c,n} = d_{c,n} \cdot (\sum_{i=1}^{C} d_{i,n})^{-1} \tag{4.6}$$

Tf is a *local* weighting, because it only consider the single document. A *global* weighting, instead, normalizes documents' characteristics by taking into account the entire corpus. Among the many forms of global weighting proposed by researchers, the mostly used one is called *inverse document frequency* (idf):

$$idf_c = \log|D| - \log|\{d : t_c \in d\}| + 1 \tag{4.7}$$

In the *idf* formula, the more common a term is among all the documents, the less (in a logarithmic fashion) it is weighted. The result is that very common terms (*e.g.*, `ArgoUML` in the homonymous email set) have an extremely low weight, because they do not help in distinguishing documents.

We follow the state of the art [3; 126] and adopt the composition of term frequency and inverse document frequency, known as *tf-idf* and defined as:

$$tf\text{-}idf_{c,n} = tf_{c,n} \cdot idf_c \tag{4.8}$$

Query Creation. Once the matrix is created, we can evaluate textual queries on it. Each query is handled as a new document and is converted to a term vector (q), by using the aforementioned weighting scheme (it only considers terms already in the TDM). Once the query vector (q) is obtained, it is compared to the vectors (d) in the TDM. The response to the query is formed by the documents corresponding to vectors that are the most "similar" to the query vector. The *similarity* is evaluated as the cosine of the angle between the vectors [28]. Any document whose distance is less than a given threshold is a match. We consider emails as the corpus, from which to create the TDM, and code artifacts as the queries. Determining the most effective threshold is non-trivial, because it depends on characteristics of the corpus and the domain [123]. When evaluating the results of IR approaches, we use a wide range of thresholds to find whether an optimal value exists.

Tokenization. The effectiveness of methods based on VSM critically relies on the quality of the term extraction process (*i.e.*, tokenization). In fact, the choice of terms determines the configuration of the TDM, the weighting values, and similarity calculations. For this reason, previous traceability recovery work in software engineering conducted a preliminary normalization phase. In particular, Marcus and Maletic, and Antoniol *et al.* deemed necessary to: (1) remove punctuation, (2) transform all the characters to their lower case versions, (3) perform stemming (*i.e.*, a morphological analysis to convert plurals into singulars and to transform conjugated forms of verbs into infinitives), and (4) remove *stop-words* (*i.e.*, very common words, such as conjunctions or prepositions, not useful to distinguish documents). The resulting terms were those considered to build the TDM.

In our experiment, we approached the pre-processing phase by first inspecting the emails in our corpus. This qualitative analysis led us to the following conclusions:

- **Punctuation:** We found that punctuation marks have to be treated differently depending on their type, usage, and location. The reason why punctuation marks are removed in most pre-processing phases (*e.g.*, [3; 126]) is that, if used *per se* without their context, they are ineffective to distinguish documents or clarify their content. In fact, in TDM the order and context of the terms is lost. Nevertheless, symbols often clarify the intended meaning of surrounding characters. As an example, the abbreviation `e.g.,` loses sense if reduced to the terms `e` and `g`, thus punctuation is necessary to preserve its meaning. In our corpus, in many cases code entities are referenced through their fully qualified names (FQN), such as `org.main.Argo`. Reducing a FQN to its components (*e.g.*, `org`, `main`, `Argo`) diminishes the quality of the information. Similarly a complete file path is more informative than the single components. Preserving this kind of punctuation assures the integrity of meaningful information. In our experiment, we consider FQNs, paths, floating numbers, abbreviations, *etc.* as single tokens.

- **Character lowercasing:** In our corpus, lowering the case of each character would not allow us to take advantage of the naming conventions (explained in Section 4.3) of each programming language. For example, a document with many occurrences of the term `Model` would more probably be referencing the homonymous class; while a document with most occurrences of the term `model`, even if also containing an occurrence of `Model` (which could be the beginning of a new sentence), would probably be discussing about the abstract concept. Our initial tests confirmed that lowercasing reduces the effectiveness of the used approaches. We thus do not lowercase characters in our experiment.

- **Stemming:** In the systems constituting our corpus, several code entities have similar names. For example, Habari has two classes called `Plugin` and `Plugins`. Performing stemming would not allow us to recognize the correct reference: The two different terms would be reduced to a single term representing both classes, thus increasing the risk of false positives. For this reason, we do not perform stemming.
- **Stop-words removal:** From the manual inspection of our corpus, we could not qualitatively understand the actual impact of removing stop words before constructing the TDM. Since there is no previous work on this, especially in our context, we evaluate the results for both cases, *i.e.*, with and without stop-words, to understand their actual impact.

4.4.2 Latent Semantic Indexing (LSI)

Documents written in natural language are inherently ambiguous, especially due to *synonymy* and *polysemy*. In our context, because of synonymy, the same entity can be referenced in an email in different ways beyond its formal identifier; e.g., ArgoUML developers often use the name `NSUML` when referring to the class `NSUMLModelFacade`. Due to polysemy, when a common dictionary word is also used as a code entity name (*e.g.*, `Model`), it is difficult to interpret its meaning without the context.

LSI aims to overcome these natural language problems by detecting the relationships between terms, modeling the *latent semantic structure* of the corpus and recognizing its *topics* [60]. LSI builds these underlying relationships by considering the words that co-occur in multiple documents. Instead of relying on individual words to locate documents, LSI uses topics: documents are no longer represented by vectors of term frequencies but by *vectors of topics*, which are inferred from the terms' co-occurrences (*e.g.*, if the terms `NSUML` and `Model` occur often together, a document containing only `NSUML` might still be returned if the query is `Model`).

LSI starts where VSM approaches stop: Given a TDM, LSI outputs a reduction through Singular Value Decomposition (SVD) [106], a technique used in signal processing to reduce noise while preserving the original signal. LSI assumes that the original TDM is filled with the noise generated by synonymy and polysemy, thus its reduction is a model of the corpus with this noise filtered out. The emails form the corpus and LSI starts from the TDM generated for the VSM approach. Once the TDM is obtained, SVD reduces pruning out eigenvectors with low values. The dimension of the resulting matrix is the number of topics to consider.

After SVD computes the reduced matrix, we can use any text as a query. As for VSM, we consider a code artifact as our query, which is transformed in a vector and compared to other documents in the matrix by using the cosine of their angles. To index the query, a naive, but slow, approach would be recomputing the entire SVD matrix with the query document added to the matrix and extract its vector. Instead, we use topic inference techniques to recover the topic composition of the query document based on its term frequency [29].

4.4.3 Results

We explore the impact of different settings for the parameters of VSM and LSI: (1) discussing the impact of topics for LSI; (2) exploring the various query types for both approaches; (3) measuring the best distance thresholds; and (4) investigating the role of corpus size. We performed an exhaustive comparison of all parameter combinations, but here we outline only general trends.

Impact of the number of topics. The number of topics impacts both the results of the approach and the time for corpus indexing and query comparison. The quality of the results and the computation time increase with the number of topics. However, the quality decreases when the number of topics exceeds a certain threshold. We are interested in finding the minimal, but still effective, number of topics. Figure 4.7 plots the best F-Measure values obtained with LSI, by number of topics and query type. We see a performance plateau after 200 topics, and a maximum around 250 topics.

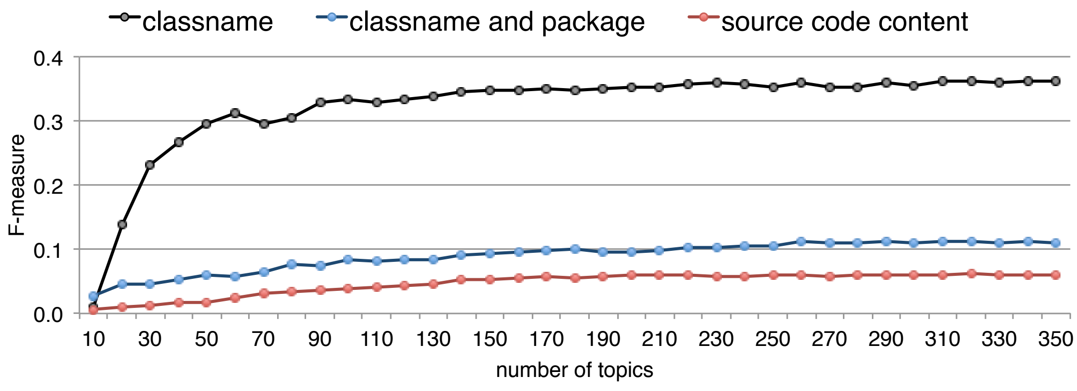


Figure 4.7: LSI, F-Measure by topics and query type

Most effective query type. We considered three different kind of queries generated from the entity to search links for: (1) the entity name; (2) the entity name and the package in which it resides; (3) the whole source code of the entity. The entity name query is the best performing, while the others provide too much noise (Figure 4.7). We obtained consistent results using VSM with tf-idf.

Optimal distances. Considering the F-Measure as the indicator of overall performance, in Figure 4.8 we see that the best distance threshold for VSM with tf-idf is in the 0.85–0.91 range.

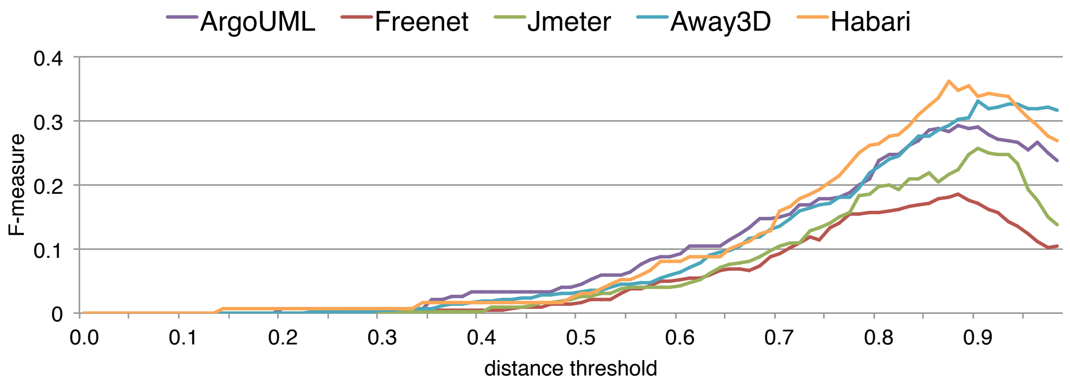


Figure 4.8: VSM, tf-idf: F-Measure by distance

For LSI the results are less conclusive (Figure 4.9): The optimal distance is heavily dependent on the system (*e.g.*, for Freenet is in the 0.25–0.35 range, while 0.6–0.8 for Habari and 0.9–1 for Augeas), thus one must discover the optimal distance for each case.

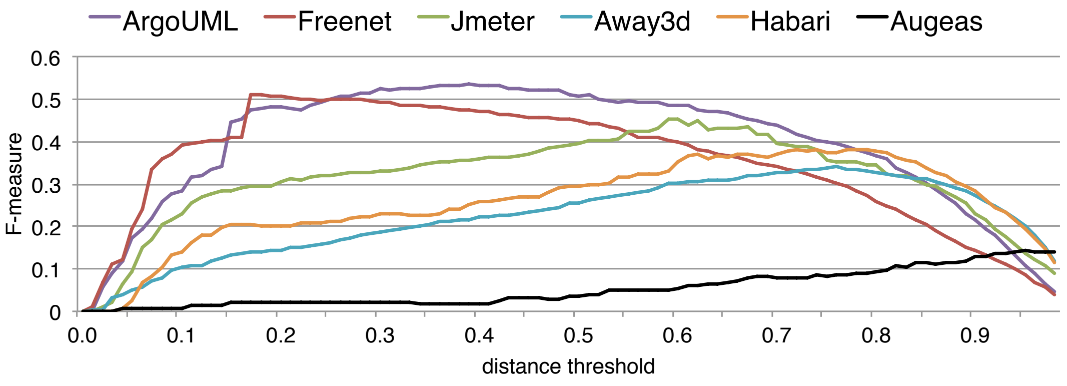


Figure 4.9: LSI: F-Measure by distance

Use of full/partial corpus. We computed the results detailed above using only the manually annotated emails included in the benchmark as the corpus. However, unlike regular expressions, IR techniques are affected by the corpus used. A larger corpus might significantly change weightings of VSM with tf-idf, and alter topics inferred by LSI. On the one hand, the topic descriptions and weightings could be more accurate, on the other hand there might be much more noise. To test this, we used VSM and LSI to index the entire mailing list content and have a full corpus. Then, when running the benchmark, we kept only the documents also in the benchmark as valid results, discarding the others. Our tests show that using a full corpus has a harmful effect for both approaches: VSM's results are much lower, while LSI's performance seems to improve only with a very large number of topics. Unfortunately, large numbers of topics (3,000 or more) are very expensive to compute when generating the approximate matrix and also increase the time needed for the distance computation, i.e., it took more than 24 hours[2] to obtain the approximate matrix from a complete mailing list using a fast C implementation of SVD[3]. Linking a single class to the same complete mailing list using lightweight approaches takes seconds. Moreover, results are worse than with a restricted corpus: with 100 topics the maximum F-Measure value reached is 0.06, 0.19 with 1,000 topics, and 0.24 with 3,000 topics.

Overall results. Table 4.3 shows the overall results with optimal parameters (entity name as a query type, best overall distance, restricted corpus, 200-300 topics for LSI). As expected, IR methods achieve higher recall values than lightweight methods, but at a significant cost in precision. F-Measure values for LSI outperform VSM with tf-idf results, probably because its ability to deal with synonyms, but are still far from the performance of the lightweight methods.

Both the techniques applied on Augeas still offer poor results. Before applying the techniques on all the systems we pre-processed the text—converting it to lowercase—, thus case sensitivity cannot be the cause. However, many components (such as executables or configuration settings)

2 on a dual quad-core Intel Xeon server with 42GB of RAM
3 http://tedlab.mit.edu/ dr/SVDLIBC/

Table 4.3: VSM and LSI results, optimal parameters

System	Vector Space Model, tf-idf			Latent Semantic Indexing		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
ArgoUML	0.25	0.34	0.29	0.60	0.48	0.53
Freenet	0.15	0.25	0.19	0.62	0.43	0.51
JMeter	0.21	0.34	0.26	0.52	0.40	0.45
Away3D	0.35	0.31	0.33	0.35	0.33	0.34
Habari	0.34	0.39	0.36	0.36	0.41	0.38
Augeas	0.10	0.20	0.14	0.10	0.28	0.14

have names identical to source code entities (functions and structures); they can be distinguished only by understanding the context in which they are mentioned.

4.5 Lightweight vs Heavyweight

Given the results achieved by lightweight techniques in the first part of this study (see Section 4.3), we are interested to see how they compare to the IR methods previously tested and how they perform with different OSS systems and programming languages.

4.5.1 Results of Lightweight Methods

To compare our lightweight techniques (see Section 4.3) to the state of the art, we apply them to the same benchmark (*i.e.*, B2) that we used to evaluate the latter. Table 4.4 reports the results. We only focused on the techniques that performed better in the first experiment.

Table 4.4: Results for best performing lightweight approaches

System	Artifact Name, case sensitive			Mixed, with regular expression			Mixed, with new regular expression		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
ArgoUML	0.27	0.68	0.38	0.64	0.61	0.63	0.35	0.68	0.46
Freenet	0.17	0.70	0.27	0.59	0.59	0.59	0.27	0.69	0.39
JMeter	0.15	0.73	0.25	0.59	0.65	0.62	0.30	0.72	0.42
Away3D	0.32	0.74	0.44	0.40	0.54	0.46	0.41	0.72	0.52
Habari	0.40	0.41	0.41	0.83	0.09	0.17	0.49	0.38	0.43
Augeas	0.09	0.72	0.15	0.14	0.02	0.04	0.15	0.64	0.24

Artifact Name, case sensitive. The results achieved for all the object-oriented systems are similar, as shown in columns 2–4 of Table 4.4. We obtained a lower precision with FREENET and JMETER, because they have a higher number of class names that are dictionary words (*e.g.*, Node or Client for FREENET, Cut or Copy for JMETER). We expected low performances for AUGEAS: since it is written in C, names are lowercase both for functions and structures, thus the performances are consistent with those achieved for the other systems when not using case sensitivity.

Mixed, with regular expression. Columns 5–7 of Table 4.4 shows that the results we previously obtained on a single Java system are still valid in other unrelated Java systems: both Freenet and JMeter reach a precision of 0.59 and a recall which is the same or higher.

The algorithm performs well with Away3D, reaching a F-Measure value of 0.46. However, characteristics of this system make the results for the precision lower that in it is for the Java systems: 1. due to its rapid evolution many classes are often moved between packages. Our algorithm uses the package information only for non-compounded words. In the other cases, all the possible classes with the same name, but in different packages, are returned by the algorithm; 2. in ActionScript, the programming language in which Away3D is written, it is less common to mention a class using its package.

Our algorithm applied on Augeas offers poor performances, because the C language does not follow the camel casing convention and does not have packages. The results on Habari are also low, this is due to two intrinsic characteristics of the system: 1. The majority of class names are not compounded words, so the algorithm switches always to the strict matching (which lowers the recall); 2. Namespaces (*i.e.*, packages) where introduced in PHP 5.3, and Habari developers do not use them in the considered releases, making the first part of the regular expression useless.

Mixed, with new regular expression. We reach better results for non-Java systems using the following regular expression:

```
(.*)
(:punct:|\s)+
<Entity name>
(:punct:|\s)+
(.*)
```

The intuition behind it is that an entity name is written as a single word separated from others by empty space or connected to them through source code tokens (*i.e.*, punctuation). For example, let us consider the following text:

```
1  var data:Plane
2  Casting is not necessary in SmallTalk
```

Figure 4.10: Example text with code artifact mentioned.

Line 1 shows the declaration of the variable `data` in Away3D. Although `Data` is one of the entities available, this approach does not report a link, because of case sensitivity. `Plane` however is correctly matched. We: check case sensitivity; count the capital letters (1); and use the above regular expression, which reports `Plane` as a matching entity, since it is surrounded only by punctuation and spaces. In line 2 the name of the entity `Cast` partially matches the word `Casting`. Due to the single capital letter, the regular expression is used, and it refuses the match because of the letter `i` after the last letter `t`.

This simple variation in our approach gives higher outcomes for all non-Java systems, as shown in columns 8–10 of Table 4.4. Augeas increases both recall and precision, while Away3D and Habari have a higher F-Measure due to increasing recall.

4.5.2 Comparison

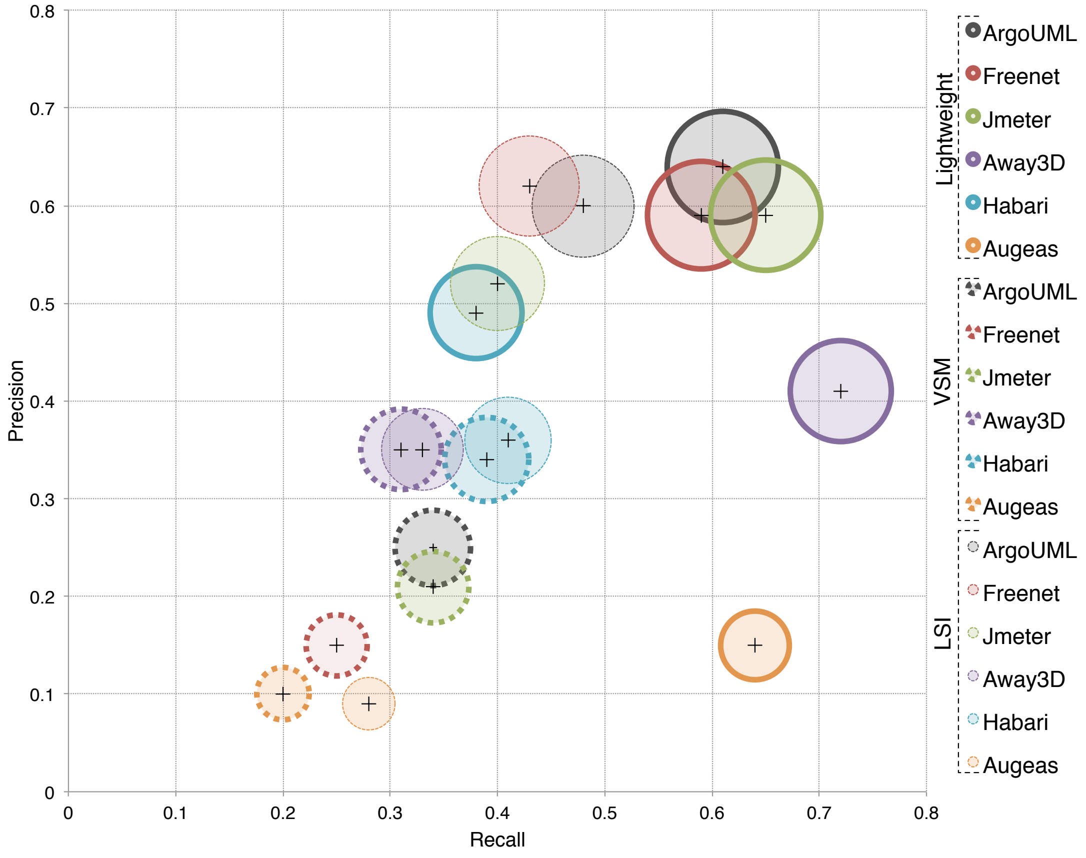


Figure 4.11: Overall precision, recall, and F-measure

Figure 4.11 summarizes the bests results obtained by all approaches. The crosses plotted on the graph represent precision and recall of each approach, while the areas of the bubbles are proportional to the F-Measure. Bubble borders differentiate the approaches: Full for the lightweight approach, thick dashes for VSM with tf-idf, and thin dashes for LSI. F-measure ranges are: 0.24–0.63 for regular expressions (choosing the regular expression according to the language of the system); 0.14–0.33 for VSM with tf-idf; and 0.14–0.53 for LSI. As it shown in graph, the lightweight methods based on regular expressions outperform information retrieval approaches consistently. Indeed, authors of emails often mention source code entities by name, hence the benefit of accounting for indirect references (the higher recall of LSI and VSM), is offset by their sensibility to noise (much lower precision). The ranking of the approaches is stable between different projects: for example, if we consider AUGEAS, which has low values for all the rankings, the lightweight approach is still the best performer, followed by LSI, and finally by VSM. This order is preserved when considering all the systems.

4.5.3 Discussion

Several additional aspects must be taken into account before drawing conclusions on our results. We discuss each approach individually, before issuing our overall recommendation.

Lightweight approaches. Our lightweight approaches are more language-dependent (with respect to other techniques): Our first mixed approach reached equivalent results on all the three Java systems, however the results for other systems are relatively more distant. This is caused by the "strict part" of the lightweight approach, *i.e.*, the regular expression, since it heavily relies on common conventions and intrinsic syntactical characteristics of the programming language. To overcome this issue, it is necessary to devise appropriate regular expressions capable of taking into account the syntactic features of the programming language of the system for which links must be found. We showed how, with a simple change, it is possible to achieve good results also in non-Java systems. However, when naming conventions are not followed and entities are not mentioned by name, this technique offers poor results, as shown by the outcome on the Augeas system, developed in C.

VSM. The VSM with tf-idf approach does not reach high values even considering the best outcome for each system. It also suffers of serious performance issues when used in very large corpuses, since it must store all the vocabulary of the corpus in the term-document matrix. For example, performance seriously decreased when we used it on the entire email population of JMeter (20,554 emails) both to build the *tdm* and to compute distances between vectors.

LSI. Theoretically, LSI should not suffer from performance issues when used in large corpuses as it reduces the size of the matrix to the approximate one, which has a lower rank. However, in practice, computing the approximate matrix of a very large corpus is very expensive. If using the same number of topics used in a small corpus, the results are not maintained. We measured how, for source code to emails linking, it is necessary to increase the number of topics when having a larger corpus to improve results. Even impractical number of topics (computing 3,000 topics took more than 24 hours) did not provide good results when the entire mailing list was indexed. For this reason, LSI suffers from the same scalability problems as VSM. This issue was also reported in other applications of LSI to standard IR corpuses [61].

Hence, our final recommendation is that the IR approaches we tried are too heavyweight and still not accurate enough to be worth the investment. In addition, they do not help to solve the problem that code entities are often referred to in ways other than their actual names. The best approach to link email and source code is using regular expressions, while being careful that these are tailored to the programming language in use.

4.5.4 On The Threats to Validity

Construct Validity. Threats to construct validity are concerned with whether what one measures is what one intends to measure. In our case, there could be several reasons why the links established between the emails and the source code as part of the benchmark are incorrect. We rely on human judgment to annotate the emails, which is a potentially error-prone process. To alleviate this issue, 50% of the emails we inspected were annotated by two different judges. When measuring the agreement between them, we found an overlap of 92%, where the 8% of disagreement was due to one judge missing one link in some emails. We corrected these errors in the set of email that was inspected twice. We expect the same low proportion of missing links in the other half of the sample, which may affect the accuracy of the results.

Another issue is the domain knowledge of the judges. Being unfamiliar with the reviewed systems, they may miss some implicit references (*e.g.*, abbreviations) to entities that a seasoned developer of the system might understand. A qualitative evaluation of our benchmark that involves system developers could measure the effect of this threat.

Statistical Conclusion. Threats to statistical conclusion are concerned with whether we have enough data to support our claims with a reasonable confidence. We took samples of email populations that were representative with a 95% confidence and a 5% error level, which are standard values. On the number of links, our corpus has 2,749 mail-to-code links, about 20 times as many as in Antoniol's study [3].

External Validity. Threats to external validity are concerned with the generalizability of the results. Our first experiment has an important limitation caused by the fact that it was performed considering only one system: ArgoUML. The style of the discussions in different mailing lists can vary, potentially changing the results achieved. However, in our approaches we did not rely on the specific dialect of the ArgoUML developers to avoid reducing the generality of our results (*e.g.*, we did not match *"NSUML"* directly to the class `NSUMLModelFacade`). On the contrary, we exploited naming conventions that are common in all the Java systems.

In the second experiment, we used a benchmark (B2) that includes different programming languages and naming conventions and we compared our best methods to the state of the art in traceability. To alleviate the generalizability issues of the first experiment, we also chose 6 systems with unrelated characteristics. The systems are developed from separate communities and are implemented in 4 different programming languages in two paradigms, object-oriented and procedural. The sizes of the systems, and of their mailing lists both varied by one order of magnitude. In general, we found that if approach A performs better than approach B on a system, it tends to perform similarly on all the systems. There is one caveat: Lightweight approaches based on regular expressions are language-specific.

There are however some aspects in which our selection is not representative: We only consider open-source systems. Usage patterns may vary in other languages (*e.g.*, scripting languages) and in the industry. In particular, mailing lists often occupy a central role in the development of open-source systems, which may not be the case in systems developed in a more centralized and confidential fashion. Finally, we have not analyzed truly large-scale systems (our largest system has around 2,000 classes): we cannot confirm that our results are similar in these cases. In particular, we expect the VSM and LSI approaches to become more resource-intensive as systems and email sets grow in size.

4.6 Related work

According to Zhao *et al.* [209], the approaches dealing with the problem of traceability between source code and unstructured data can be classified in three categories: (1) artifact traceability support tools, (2) traceability via intermediate abstraction, and (3) traceability via IR.

Artifact Traceability Support Tools

CASE (Computer-Aided Software Engineering) tools help developers to manually maintain links between source code and other artifacts, providing support for recording, displaying and checking links. TOOR (Traceability of Object-Oriented Requirements) [156] is a visual tool intended to be used during development: A programmer can define any artifact (*e.g.*, design charts, system manuals, interview transcripts) as an object of a certain class and establish relations to other artifacts. Objects and their relationships can be inspected graphically. For instance, all objects of a given class, or all objects participating in a given relation, can be shown at the same time. The tool can also relate indirectly linked objects, as the links are transitive.

The REMAP [159] and gIBIS [47] tools are based on IBIS (Issue Based Information System) [205], a method that provides a model for representing "argumentation" processes, and hence keeping the rationale for the decisions. Adams is a semi-automated traceability link recovery tool based on LSI and is implemented as a plug-in for Eclipse [59].

The Ophelia Traceability Layer [180] is a tool that takes into account all the artifacts of the development lifecycle, helping developers to create a navigable graph of their relationships, to maintain the traceability between artifacts consistently.

Artifact Traceability via Intermediate Abstraction

These methods rely on an intermediate abstraction, for both source code and documents, as a basis for the matching. While we strive for investigating methods to *automatically* maintain links, these methods require user interaction. For this reason we do not cover them in detail.

Fiutem and Antoniol [70] and Antoniol *et al.* [4] proposed an Abstract Object Language (AOL) to assist the traceability between code and design artifacts. Murphy *et al.* [139] introduced reflexion models for the same purposes, while Sefika *et al.* [177] proposes a model based on both static and dynamic information.

Artifact Traceability via Information Retrieval

Probabilistic and Vector Space Model (VSM). Antoniol *et al.* experimented with two Information Retrieval (IR) models to retrieve the link between source code and documentation [3]: a *Probabilistic IR Model* and a *Vector Space IR Model*. The former computes a ranking score based on the probability that a document is related to a specific source code component, while the latter calculates the distance between the vocabulary of all the documents and a code component. They analyzed two software systems: LEDA (Library of Efficient Data Types and Algorithms), a C++ library of 208 classes to link to 88 manual pages, and a student project of 95 classes to link to functional requirements written beforehand.

Antoniol *et al.* used the document collection as the corpus in which to find the missing links, and source code files as the queries. Each document was pre-processed: The authors converted characters to lowercase, removed stop-words (*e.g.*, articles, punctuation, numbers, common words), and performed stemming (*e.g.*, converting plurals into singulars, transforming verbs into their infinitive form). They also extracted the list of identifiers from the source code, removed the language keywords, and split compounded words (*e.g.*, *ClassName* into *class name*). Code comments were disregarded. Both case studies revealed a better overall performance of VSM.

Latent Semantic Indexing (LSI). Marcus and Maletic proposed a solution based on LSI [126]. LSI is based on VSM and considers that words always appear in a context, which provides a set of mutual constraints to determine meaning similarity. The authors evaluated their approach effectiveness on the systems considered by Antoniol *et al.* [3].

Marcus and Maletic considered source code files as the corpus in which to find the link, and documents as the queries. The pre-processing consisted in: non-textual tokens removal, characters lowercasing, and compounded words splitting (while also keeping the original form, *i.e.*, *ClassName* becomes *classname*, *class*, *name*). They included code comments. Finally, as LSI does not use a predefined vocabulary or a predefined grammar, the authors deemed not necessary to perform the stemming process, *i.e.*, there was no morphological analysis. The results were slightly improved with respect to the approach by Antoniol *et al.*, especially for the LEDA system: There were more documents in the corpus and the same entity identifiers were used in both the source code and the documents.

Hayes *et al.* used the three IR algorithms proposed by Antoniol *et al.* and Marcus *et al.* (Vector Space Model, Vector Space Model with a simple thesaurus, and LSI) to trace requirements-to-requirements and aggregate candidate links to be evaluated by software analysts. They validated the algorithms on two systems of similar size to the ones used by Antoniol *et al.* (one of circa 20 kLOC of C and 455 documents, and the other with 58 documents).

Lormans *et al.* used LSI to find traceability relations between requirements, design documents, and test cases [120]. They evaluated the effectiveness of LSI in terms of precision and recall on two compact case studies.

Chen and Grundy incorporated regular expressions, key phrases [202], and a modified version of the K-mean clustering algorithm [122] with VSM to extract links between documents and class entities [45]. They show that the three techniques significantly improve—although by a low magnitude—the on shortcomings of VSM by taking advantage of the respective strengths of each of the three supporting techniques.

Latent Dirichlet Allocation (LDA) and Jensen–Shannon divergence Other than the aforementioned technique, also Jensen-Shannon divergence [51] and Latent Dirichlet Allocation (LDA) [39] were used for traceability link recovery. Specifically, Asuncion *et al.* propose an approach based on LDA to aid users in analyzing the semantic nature of artifacts and the software architecture [7]. Abadi *et al.* reported good results in using the Jensen-Shannon model for the traceability discovery task [1].

Natural Language Processing (NLP). Baysal *et al.* put discussion archives (*i.e.*, the emails in mailing lists) and source code [23] in correlation. They looked for a correlation between discussions and software releases. First, they recovered information about the system applying data mining techniques on its release history and discussion archives. Then, they used NLP methods to search for a correlation. They presented two case studies: A visualization tool (a Java system with 144 files and an archive of 495 emails) and Apache Ant (a Java system with 667 Java files and an archive of 67,377 emails). Baysal *et al.* did not manually inspect the studied systems to verify the quality of their results.

4.7 Summary

Email archives enclose significant information on the software system they discuss. We dealt with the problem of recovering traceability links between emails and source code. We evaluated different automated approaches to retrieve these links: Lightweight methods based on capturing programming languages elements with regular expressions, and two Information Retrieval approaches. We tested all approaches against the benchmark we created and measured their effectiveness in terms of precision, recall and F-measure. The results we obtained show how, for this task, “less is more”: The lightweight methods consistently and significantly outperform the IR approaches in all six systems. The reason is that in emails entities are often referred to by name, not synonyms, and source code is rare.

Reflection. We claimed that recovering traceability links between emails and code artifacts would bring several benefits to understand and support software development. However, the links are only the *first*, though necessary, step toward proving our thesis. In the following chapters of this part, we prove the usefulness of the linked email data by performing two case studies: (1) We create a new quantitative metric for code entities, which takes into account email discussions as a new source of information, (2) and we implement a recommender system to integrate emails in the development environment and use it to support program comprehension.

Chapter 5

Improving Defect Prediction Approaches With Email Data

In the previous chapter, we presented both lightweight and sophisticated techniques for reconnecting development emails to the code entities they discuss. In this chapter we show that, by having this novel information available, we can use it to conduct defect prediction analysis.

5.1 Overview

Knowing the location of future defects allows project managers to optimize the resources available for the maintenance of a software project by focusing on the problematic components. However, performing defect prediction with enough precision to produce useful results is a challenging problem. Researchers have proposed a number of approaches to predict software defects, exploiting various sources of information, such as source code metrics [21; 147; 183; 88; 142], code churn [141], process metrics extracted from versioning system repositories [137; 27], and past defects [152; 212]. A source of information for defect prediction that has been not exploited so far are development mailing lists.

Why would one want to use emails for defect prediction? The source code of software systems is only written by developers, who must follow a rigid and terse syntax to define abstractions they want to include. On the contrary, mailing lists, even those specifically devoted to development, archive emails written by both programmers and users. For this reason, the entities discussed are not only the most relevant from a development point of view, but also the most exploited during the use of a software system. Moreover, emails are written using natural language, thus the writer can generalize some concepts and focus on others. For this reason, we expect information we extract from mailing lists to be independent from those provided by the source code analysis. Thus, they can be valuable information for defect prediction.

We present "popularity" metrics that express the importance of each source code entity in discussions taking place in development mailing lists. Our hypothesis is that such metrics are an indicator of possible flaws in software components, thus being correlated with the number of defects. We aim at answering the following research questions:

Q1 – *Does the popularity of software components in discussions correlate with software defects?*

Q2 – *Is a regression model based on the popularity metrics a good predictor for software defects?*

Q3 – *Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*

We provide the answers to these questions by validating our approach on four different open source software systems.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We extend* Miler (see Section 3.2) *to model bug information*. We extend our meta-model to support novel metrics generated by our lightweight email-to-code linking approach.
- *We devise a novel defect prediction technique based on email data*. We evaluate the performance of our model on four OSS systems. We integrate our novel information to existing models for defect prediction and show that the results are improved.

Structure of the chapter. In Section 5.2 we expose the methodology used to conduct our experiment: how we collected, processed, and analyzed the data in order to construct and test popularity metrics and other metrics we include in our case study. In Section 5.3 we describe the dataset of our case study, the evaluation and the results achieved. We discuss our findings in Section 5.4. In Section 5.5 we list the possible threats to the validity of our experiment and how we tried to reduce them. We review related work in Section 5.6, and summarize our contributions in Section 5.7.

5.2 Methodology

Our goal is to investigate whether popularity metrics correlate with software defects, then to study whether existing bug prediction approaches can be improved using such metrics. To conduct our investigation, we follow the methodology depicted in Figure 5.1:

- We extract email data, link it with source code entities and compute popularity metrics. We extract and evaluate source code and change metrics.
- We extract defect data from issue repositories and we quantify the correlation of popularity metrics with software defects, using as baseline the correlation between source code metrics and software defects.
- We build regression models with popularity metrics as independent variables and the number of post-release defects as the dependent variable. We evaluate the performance of the models using the Spearman's correlation between the predicted and the reported bugs. We create regression models based on source code metrics [183; 88; 142; 212] and change metrics [137; 27] alone, and later enrich these sets of metrics with popularity metrics, to measure the improvement given by the popularity metrics.

We focus on Java software systems using classes as target entities. We decided to not focus on packages for the following reasons:

- Predictions at the package-level are less helpful since they are significantly larger than classes. Reviewing a package involves more work than reviewing a class, because the distribution of bugs across classes in a package is seldom uniform.
- Package-level information can be derived from class-level information, while the opposite is not true.

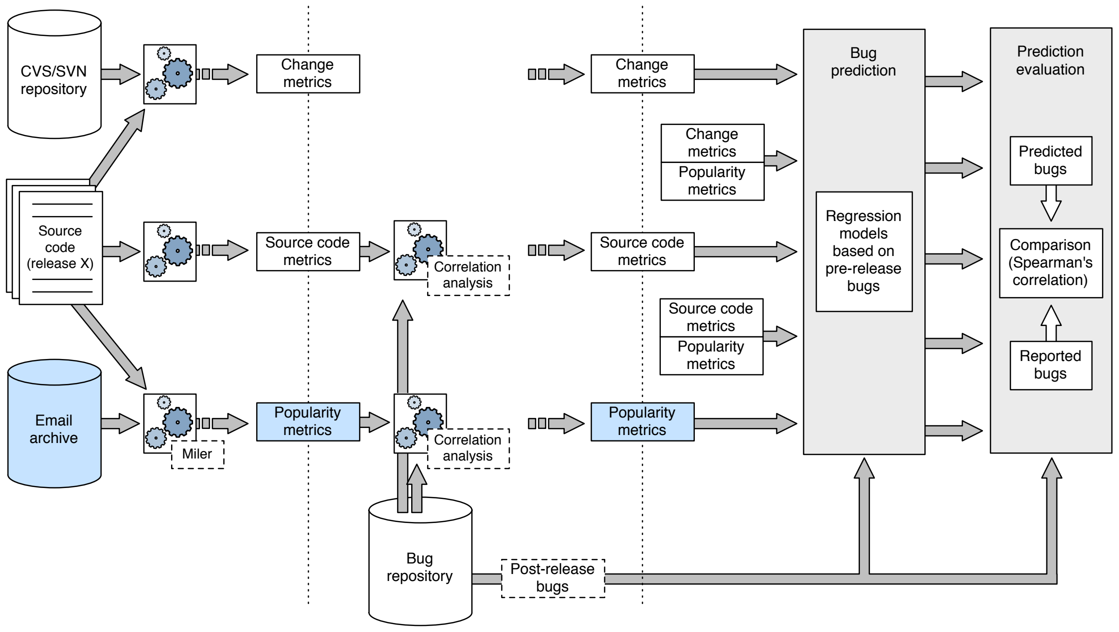


Figure 5.1: Overall schema of our approach.

- Classes are the building blocks of object-oriented systems, and are self-contained elements from the point of view of both design and implementation.

Modeling. We import into Miler the object-oriented models systems we want to analyze, by using inFusion and the Moose reengineering environment (see Section 3.2.1).

Computing Source Code Metrics. The model of a software system obtained through Moose also contains a catalogue of object oriented metrics, listed in the first four columns of Table 5.1. The catalog includes the Chidamber and Kemerer (CK) metrics suite [46], which was already used for bug prediction [21; 66; 183; 88], and additional object oriented metrics.

Computing Change Metrics. Change metrics are *process metrics* extracted from versioning system log files (CVS and SVN in our experiments). Differently from source code metrics that measure several aspects of the source code, change metrics are measures of how the code was developed over time. We use the set of change metrics listed in the last two columns of Table 5.1, which is a subset of the ones used by Moser *et al.* [137].

To use change metrics in our experiments, we need to link them with source code entities, *i.e.*, classes. We do that by comparing the versioning system filename, including the directory path, with the full class name, including the class path. Due to the file-based nature of SVN and CVS and to the fact that Java inner classes are defined in the same file as their containing class, several classes might point to the same CVS/SVN file. For this, we do not consider inner Java classes.

Table 5.1: Class level source code and change metrics.

CK Metrics	
Name	**Description**
WMC	Weighted Method Count
DIT	Depth of Inheritance Tree
RFC	Response For Class
NOC	Number of Children
CBO	Coupling Between Objects
LCOM	Lack of Cohesion in Methods

Change Metrics	
Name	**Description**
NR	Number of revisions
NFIX	Number of times file was involved in bug-fixing
CHGSET	Change set size (max and avg)
NREF	Number of times file has been refactored
NAUTH	Number of authors who committed the file
AGE	Age of a file

Other OO Metrics	
Name	**Description**
FanIn	Number of classes referencing the class
FanOut	Number of classes referenced by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
LOC	Number of lines of code
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

Computing Popularity Metrics. The extraction of popularity metrics, given a software system and its mailing lists, is done in two steps: First we link each class with all the emails discussing it, by using the lightweight techniques presented in Chapter 4, then we use the `popularity metrics extractor` (see Section 3.2.1) to compute the metrics using the obtained links.

Table 5.2: Class level email popularity metrics.

Popularity Metrics	
Name	**Description**
POP-NOM	Number of emails
POP-NOT	Number of threads
POP-NOA	Number of authors
POP-NOCM	Number of characters in email
POP-NOMT	Number of emails in thread

For each popularity metric (listed in Table 5.2) we provide the rationale behind their creation and a high-level description of their implementation, using Miler's meta-model.

POP-NOM (Number of emails): To associate the popularity of a class with discussions in mailing lists, we count the number of mails that are mentioning it. Since we are considering development mailing lists, we presume that classes are mainly mentioned in discussions about failure reporting, bug fixing and feature enhancements, thus they can be related to defects. Thanks to the enriched FAMIX model we generate, it is simple to compute this metric. Once the mapping from classes to emails is completed, and the model contains the links, we count the number of links of each class.

POP-NOT (Number of threads): In mailing lists discussions are divided in *threads*. Our hypothesis is that all the messages that form thread discuss the same topic: If an author wants to start talking about a different subject she can create a new thread. We suppose that if developers are talking about one defect in a class they will continue talking about it in the same thread. If they want to discuss about an unrelated or new defect (even in the same classes) they would open a new thread. The number of threads, then, could be a popularity metric whose value is related to the number of defects. After extracting emails from mailing lists, our email model also contains the information about threads. Once the related mails are available in the object-oriented model, we retrieve this thread information from the messages related to each class and count the number of different threads. If two, or more, emails related to the same class are part of the same thread, they are counted as one.

POP-NOA (Number of authors): A high number of authors talking about the same class suggests that it is subject to broad discussions. For example, a class frequently mentioned by different users can hide design flaws or stability problems. Also, a class discussed by many developers might be not well defined, comprehensible, or correct, thus more defect prone. For each class, we count the number of authors that wrote in referring mails (*i.e.*, if the same author wrote two, or more, emails, we count only one).

POP-NOCM (Number of characters in emails): Development mailing lists can also contain other topics than technical discussions. For example, while manually inspecting part of our dataset, we noticed that voting about whether and when to release a new version occurs quite frequently in Lucene, Maven and Jackrabbit mailing lists. Equally, announcements take place with a certain frequency. Usually this kind of messages is characterized by a short content (*e.g.*, "yes" or "no" for voting, "congratulations" for announcements). The intuition is that emails discussing flaws in the source code could present a longer amount of text than mails about other topics. We consider the length of messages taking into account the number of characters in the text of mails: We evaluate the POP-NOCM metric by adding the number of characters in all the emails related to the chosen class.

POP-NOMT (Number of emails in threads): Inspecting sample emails from the mailing lists which form our experiment, we noticed that short threads are often characteristic of "announcements" emails, simple emails about technical issues experimented by new users of the systems, or updates about the status of developers. We hypothesize that longer threads could be symptom of discussions about questions that raise the interest of the developers, such as those about defects, bugs or changes in the code. For each class in the source code, we consider the thread of all the referring mails, and we count the total number of mails in each thread. If a thread is composed by more than one email, but only one is referring the class, we still count all the emails inside the thread, since it is possible that following emails reference the same class implicitly.

Extracting Defect Information. To measure the correlation of metrics with software defects, and to perform defect prediction, we need to link each problem report to any class of the system that it affects. We link classes imported in MILER with versioning system files, as we did to compute change metrics, and the files with bugs retrieved from a BUGZILLA[1], or JIRA,[2] repository. Figure 5.2 shows the bug linking process.

A file version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (*e.g.*, "fixed bug 123"). Such references allow us to link problem reports with files in the versioning system, and therefore with source code artifacts, *i.e.*, classes. However, the link between a CVS/SVN file and a BUGZILLA/JIRA problem

1 `http://www.bugzilla.org`
2 `http://www.atlassian.com/software/jira`

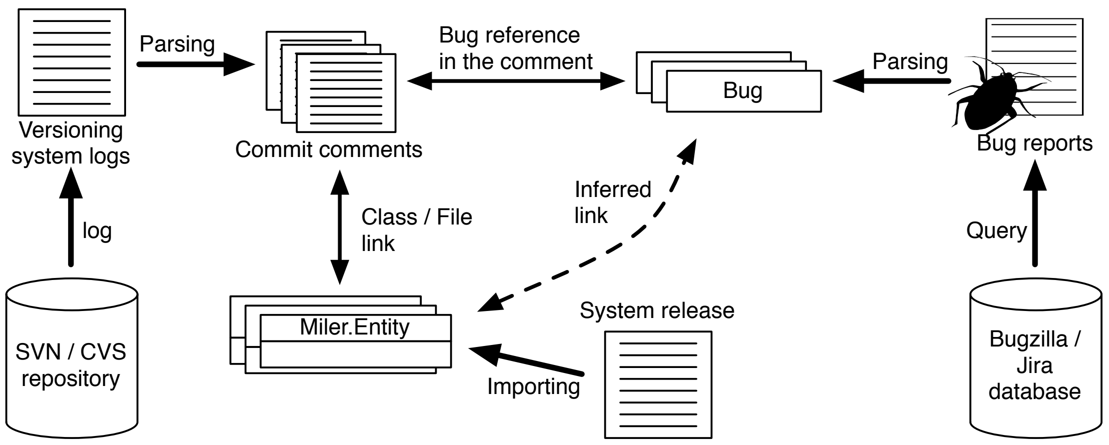


Figure 5.2: Linking bugs, SCM files and classes.

report is not formally defined, and to find a reference to the problem report id we use pattern matching techniques on the developer comments, a widely adopted technique [69; 212]. Once we have established the link between a problem report and a file version, we verify that the bug has been reported before the commit time of the file version.

To measure the correlation between metrics and defects we consider all the defects, while for bug prediction only post-release defects, *i.e.*, the ones reported within a six months time interval after the considered release of the software system.[3]. The output of the bug linking process is, for each class of the considered release, the total number of defects and of post-release defects.

Summing Up. We presented the different data repositories we mine, how we parse, model and extract all the necessary information. We described our popularity metrics, along with all the other metrics included in our experiment. In the following section we describe the dataset on which we compute these metrics and how we evaluate and compare them.

5.3 Experiments

We conducted our experiments on the software systems listed in Table 5.3. We considered systems that deal with different domains and have distinct characteristics (*e.g.*, popularity, number of classes, emails, and defects) to mitigate some of the threats to external validity. These systems are stable projects, under active development, and have a history with several major releases. All are written in Java to ensure that all the code metrics are defined identically for each system. By using the same parser, we can avoid issues due to behavior differences in parsing, a known issue for reverse engineering tools [110].

Public development mailing lists used to discuss technical issues are available for all the systems, and are separated from lists specifically thought for system user issues. We consider emails starting from the creation of each mailing list until the date of each release considered. Messages

3 Six months for post release defects was also used by Zimmermann *et al.* [212].

Table 5.3: Systems considered in the study

System	Classes	Emails		Defects	
		Total	Linked	Timeframe	Amount
Equinox	439	5,575	2,383	Feb 2003 - Jun 2008	1,554
Jackrabbit	1,913	11,901	3,358	Sep 2004 - Dec 2008	674
				Sep 2004 - Aug 2009	975
Lucene	1,279	17,537	8,800	Oct 2001 - May 2008	751
				Oct 2001 - Sep 2009	1,274
Maven	301	65,601	4,616	Apr 2004 - Sep 2008	507
				Apr 2004 - Aug 2009	616

automatically generated by bug tracking and revision control systems are filtered out, and we report the resulting number of emails and the number of those referring to classes according to our linking techniques. All systems have public bug tracking systems that were usually created along with the mailing lists.

5.3.1 Correlations Analysis

To answer the research question Q1 "*Does the popularity of software components correlate with software defects?*", we compute the correlation between class level popularity metrics and the number of defects per class. We compute the correlation in terms of both the *Pearson's* and the *Spearman's* correlation coefficient (r_{prs} and r_{spm}, respectively). The Spearman's rank correlation test is a non-parametric test that uses ranks of sample data consisting of matched pairs. The correlation coefficient varies from 1, *i.e.*, ranks are identical, to -1, *i.e.*, ranks are the opposite, where 0 indicates no correlation. Contrarily to Pearson's correlation, Spearman's one is less sensitive to bias due to outliers and does not require data to be metrically scaled or of normality assumptions [193]. Including the Pearson's correlation coefficient increases our understanding of the results: If r_{spm} is higher than r_{prs}, we might conclude that the variables are consistently correlated, but not in a linear fashion. If the two coefficients are very similar and different from zero, there is indication of a linear relationship. Finally, if the r_{prs} value is significantly higher than r_{spm}, we can deduce that there are outliers inside the dataset. This information first helps us to discover threats to construct validity, and then put in evidence single elements that are heavily related. For example, a high r_{prs} can indicate that, among the classes with the highest number of bugs, we can find also the classes with the highest number of related emails.

We compute the correlation between class level source code metrics and number of defects per class, in order to compare the correlation to a broadly used baseline. We only show the correlation for the source code metric LOC, as previous research showed that it is one of the best metrics for defect prediction [88; 150; 151; 153]. Table 5.4 shows the correlation coefficients between the different popularity metrics and the number of bugs of each system.

Table 5.4: Correlation coefficients

System	POP-NOM		POP-NOCM		POP-NOT		POP-NOTM		POP-NOA		LOC	
	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}
Equinox	0.52	0.51	0.52	0.42	**0.53**	**0.54**	0.52	0.48	0.53	0.50	0.73	0.80
Jackrabbit	0.23	0.35	0.22	0.36	**0.24**	**0.36**	0.23	-0.02	0.23	0.34	0.27	0.54
Lucene	0.41	0.63	0.38	0.57	0.41	0.57	**0.42**	**0.68**	0.41	0.54	0.17	0.38
Maven	0.44	0.81	0.39	0.78	**0.46**	0.78	0.44	**0.81**	0.45	0.78	0.35	0.78

We put in bold the highest values achieved for both r_{spm} and r_{prs}, by system. Results provide evidence that the two metrics are rank correlated, and correlations over 0.4 are considered to be strong in fault prediction studies [210]. The Spearman's correlation coefficients in our study exceed this value for three systems, *i.e.*, EQUINOX, LUCENE, and MAVEN. In the case of JACKRABBIT, the maximum coefficient is 0.24, which is similar to the value reached by using LOC. The best performing popularity metric depends on the software system: for example in LUCENE, POP-NOTM, which counts the length of threads containing emails about the classes, is the best choice, while POP-NOT, number of threads containing at least one email about the classes, is the best performing for other systems.

5.3.2 Defect Prediction

To answer the research question Q2 "*Is a regression model based on the popularity metrics a good predictor for software defects?*", we create and evaluate regression models in which the independent variables are the class level popularity metrics, while the dependent variable is the number of post-release defects per class. We create regression models based on source code metrics and change metrics alone, as well as models in which these metrics are enriched with popularity metrics, where the dependent variable is always the number of post-release defects per class.

Subsequently, we compare the prediction performances of these models to answer research question Q3 "*Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*" We follow the methodology proposed by Nagappan *et al.* [142] and also used by Zimmermann *et al.* [210], consisting of: Principal component analysis, building regression models, evaluating explanative power and evaluating prediction power.

Principal Component Analysis. This method is a standard statistical technique that avoids the problem of multicollinearity among the independent variables. This problem comes from intercorrelations amongst these variables and can lead to an inflated variance in the estimation of the dependent variable. We do not build the regression models using the actual variables as independent variables, but instead we use sets of principal components (PCs). PCs are independent and do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible. We select sets of PCs that account for a cumulative sample variance of at least 95%.

Building Regression Models. To evaluate the predictive power of the regression models we do cross-validation: We use 90% of the dataset, *i.e.*, 90% of the classes (training set), to build the prediction model, and the remaining 10% of the dataset (validation set) to evaluate the efficacy of the built model. For each model we perform 50 "folds", *i.e.*, we create 50 random 90%-10% splits of the data.

Evaluating Explanative Power. To evaluate the explanative power of the regression models we use the adjusted R^2 coefficient. The (non-adjusted) R^2 is the ratio of the regression sum of squares to the total sum of squares. R^2 ranges from 0 to 1, and the higher the value is, the more variability is explained by the model, *i.e.*, the better the explanative power of the model is. The adjusted R^2, takes into account the degrees of freedom of the independent variables and the sample population. As a consequence, it is consistently lower than R^2. When reporting results, we only mention the adjusted R^2. We test the statistical significance of the regression models using the F-test. All our regression models are significant at the 99% level ($p < 0.01$).

Evaluating Prediction Power. To evaluate the predictive power of the regression models, we compute Spearman's correlation between the predicted number of post-release defects and the actual number. This approach has been broadly used to assess the predictive power of a number of predictors [150; 151; 153]. In the cross-validation, for each random split, we use the training set (90% of the dataset) to build the regression model, and then we apply the obtained model on the validation set (10% of the dataset), producing for each class the predicted number of post-release defects. Then, to evaluate the performance of the performed prediction, we compute Spearman's correlation, on the validation set, between the lists of classes ranked according to the predicted and actual number of post-release defects. Since we perform 50 folds cross-validation, the final values of the Spearman's correlation and adjusted R^2 are averages over 50 folds.

Results. Table 5.5 displays the results we obtained for the defect prediction, considering both R^2 adjusted values and Spearman's correlation coefficients.

The first row shows the results achieved using all the popularity metrics defined in Section 5.2. In the following four blocks, we report the prediction results obtained through the source code and change metrics, first alone, then by incorporating each single popularity metric, and finally incorporating all the popularity metrics. For each system and block of metrics, when popularity metrics augment the results of other metrics, we put in bold the highest value reached.

Analyzing the results of the sole popularity metrics, we notice that, in terms of correlation, Equinox and Maven still present a strong correlation, *i.e.*, higher than .40, while Lucene is less correlated. The popularity metrics alone are not sufficient for performing predictions in the Jackrabbit system. Looking at the results obtained by using all the metrics, we first note that Jackrabbit's results are much lower if compared to those reached in other systems, especially for the R^2, and partly for the R_{spm}. Only change metrics show fine R_{spm} value in Jackrabbit.

Going back to the other systems, the R^2 adjusted values are always increased and the best results are achieved when using all popularity metrics together. The increase, with respect to the other metrics, varies from 2%, when other metrics already reach high values, up to 107%. Spearman's coefficients also increase by using the information given by popularity metrics: Their values augment, on average, more than fifteen percent. However, there is not a single popularity metric that outperforms the others, and their union does not always give the best results.

Table 5.5: Defect prediction results

Metrics	R^2 *adj*					$R_{spearman}$				
	Equinox	Jackrabbit	Lucene	Maven	*Avg*	Equinox	Jackrabbit	Lucene	Maven	*Avg*
All Popularity Metrics	0.23	0.00	0.31	0.55	*0.27*	0.43	0.04	0.27	0.52	*0.32*
All Change Metrics	0.55	0.06	0.43	0.71	*0.44*	0.54	0.30	0.36	0.62	*0.45*
All C.M. + POP-NOM	0.56	0.06	0.43	0.71	*0.44*	0.53	**0.32**	0.38	**0.69**	***0.48***
All C.M. + POP-NOCM	0.58	0.06	0.43	0.70	*0.44*	0.57	0.31	**0.43**	0.60	*0.48*
All C.M. + POP-NOT	0.56	0.06	0.43	0.71	*0.44*	0.54	0.31	0.39	0.59	*0.46*
All C.M. + POP-NOMT	0.56	0.06	0.43	0.70	*0.44*	0.53	0.29	0.41	0.60	*0.46*
All C.M. + POP-NOA	0.56	0.06	0.43	0.70	*0.44*	**0.58**	0.29	0.37	0.43	*0.42*
All C.M. + All POP	**0.61**	**0.06**	**0.45**	**0.71**	***0.46***	0.52	0.30	0.38	0.43	*0.41*
Improvement	*11%*	*0%*	*+5%*	*0%*	*+4%*	*+7%*	*+7%*	*+19%*	*+11%*	*+11%*
Source Code Metrics	0.61	0.03	0.27	0.42	*0.33*	0.51	0.17	0.31	0.52	*0.38*
S.C.M. + POP-NOM	0.62	0.03	0.33	0.59	*0.39*	0.53	0.14	0.35	0.52	*0.38*
S.C.M. + POP-NOCM	0.62	**0.04**	0.32	0.56	*0.38*	0.51	0.15	0.36	0.60	*0.41*
S.C.M. + POP-NOT	0.61	0.03	0.31	0.57	*0.38*	0.49	0.15	**0.38**	0.52	*0.38*
S.C.M. + POP-NOMT	0.62	0.03	0.35	0.60	*0.40*	0.55	0.14	0.33	0.43	*0.36*
S.C.M. + POP-NOA	0.61	0.04	0.30	0.56	*0.38*	0.53	0.12	0.38	**0.70**	***0.43***
S.C.M. + All POP	**0.62**	0.03	**0.37**	**0.61**	***0.41***	**0.58**	0.14	0.32	0.52	*0.39*
Improvement	*+2%*	*+25%*	*+37%*	*+45%*	*+27%*	*+14%*	*-0.12*	*+23%*	*+35%*	*+15%*
CK Metrics	0.54	0.01	0.39	0.28	*0.31*	0.51	0.13	0.36	0.60	*0.40*
CK + POP-NOM	0.56	0.02	0.40	0.54	*0.38*	0.48	0.13	0.35	**0.69**	***0.41***
CK + POP-NOCM	0.57	0.02	0.40	0.50	*0.37*	0.50	**0.17**	0.33	0.42	*0.35*
CK + POP-NOT	0.56	0.01	0.40	0.51	*0.37*	**0.53**	0.13	0.34	0.52	*0.38*
CK + POP-NOMT	0.57	0.01	0.40	0.56	*0.39*	0.52	0.14	0.25	0.49	*0.35*
CK + POP-NOA	0.56	0.02	0.40	0.51	*0.37*	0.52	0.14	**0.41**	0.53	*0.40*
CK + All POP	**0.57**	**0.02**	**0.42**	**0.58**	***0.40***	0.51	0.16	0.30	0.52	*0.37*
Improvement	*+6%*	*+50%*	*+8%*	*+107%*	*+43%*	*+4%*	*+31%*	*+14%*	*+15%*	*+16%*
All Source Code Metrics	0.66	0.04	0.44	0.45	*0.40*	0.48	0.15	0.35	0.36	*0.33*
All S.C.M. + POP-NOM	0.67	0.04	0.45	0.60	*0.44*	**0.59**	0.15	0.34	**0.62**	***0.43***
All S.C.M. + POP-NOCM	0.66	0.04	0.45	0.56	*0.43*	0.51	0.16	0.30	0.31	*0.32*
All S.C.M. + POP-NOT	0.66	0.04	0.44	0.57	*0.43*	0.50	0.14	**0.35**	0.52	*0.38*
All S.C.M. + POP-NOMT	0.67	0.04	0.44	0.62	*0.44*	0.53	0.14	0.35	0.34	*0.34*
All S.C.M. + POP-NOA	0.66	0.04	0.44	0.57	*0.43*	0.51	0.15	0.34	0.43	*0.36*
All S.C.M. + All POP	**0.67**	0.04	**0.46**	**0.63**	***0.45***	0.51	**0.16**	0.33	0.52	*0.38*
Improvement	*+2%*	*0%*	*+5%*	*+40%*	*+12%*	*+23%*	*+7%*	*+0%*	*+72%*	*+26%*

5.4 Discussion

Popularity of software components correlates with software defects. Three software systems out of four show a strong rank correlation, *i.e.*, coefficients ranging from .42 to .53, between defects of software components and their popularity in email discussions. Only JACKRABBIT is less rank correlated with a coefficient of .23.

Popularity can predict software defects, but without major improvements over previously established techniques. In the second part of our results, consistently with the correlation analysis, the quality of predictions done by JACKRABBIT using popularity metrics are extremely low, both for the R^2 adjusted values and for the Spearman's correlation coefficients. On the contrary, our popularity metrics applied to the other three systems lead to different results: Popularity metrics are able to predict defects. However, if used alone, they do not compete with the results obtained through other metrics. The best average results are shown by the Change Metrics, corroborating previous works stating the quality of such predictors [137; 27].

Popularity metrics can improve prediction performances of existing defect prediction techniques. The most interesting results are obtained integrating the popularity information into other techniques. This, in most cases, increase the overall results: The improvements on correlation coefficients are, on average, more than fifteen percent higher, with peaks over 30% and reaching the top value of 72%, to those obtained without popularity metrics. This provides evidence toward our initial assumption that popularity metrics measure a different aspect of the development process from those captured by other techniques. Our results put in evidence that, given the considerable difference of the prediction performance across different software projects, bug prediction techniques that exploit popularity metrics should not be applied in a "black box" way. As suggested by Nagappan *et al.* [142], the prediction approach should be first validated on the history of a software project, to see which metrics work best for predictions for the system.

The case of Jackrabbit shows that all the information that we have available cannot be used to explain its defects. This underlines the importance of triangulating the investigated phenomena (*e.g.*, defects) with data gathered from different sources and that we have to adapt our approaches to each software system under study.

5.5 Threats to validity

Threats to construct validity. A first construct validity threat concerns the way we link bugs with versioning system files and subsequently with classes. In fact, the pattern matching technique we use to detect bug references in commit comments does not guarantee that all the links are retrieved. We also made the assumption that commit comments do contain bug fixing information, which limits the application of our bug linking approach only to software projects where this convention is used. Finally, a commit that is referring to a bug can also contain modifications to files that are unrelated to the bug. However, this technique is the current state of the art in linking bugs to versioning system files, widely used in literature [69]. The noise affecting issue repositories constitutes another construct validity threat: Even though we carefully removed all the issue reports not marked as *bug* (*e.g.*, *New Feature*, *Improvement*, *Task*) from our dataset, Antoniol *et al.* showed that a relevant fraction of issues marked as *bugs* in Bugzilla (according to their severity) are not actually related to corrective maintenance [2]. Another threat concerns the procedure for linking emails to discussed classes. We use linking techniques whose effectiveness was measured (see Chapter 4), and it is known that they cannot produce a perfect linking. The enriched object-oriented model can contain wrongly reported links or miss connections that are present. We alleviated this problem manually inspecting all the classes that showed an uncommon number of links, *i.e.*, outliers, and, whenever necessary, adjusted the regular expressions composing the linking techniques to correctly handle such unexpected situations. We removed from our dataset any email automatically generated by the bug tracking system and the revision control system, because they could bias the results.

Threats to statistical conclusion validity. We strived to reduce threats to statistical conclusion validity by having all the Spearman correlation coefficients and all the regression models significant at the 99% level.

Threats to external validity. In our approach we analyze only open-source software projects, however the development in industrial environment may differ and conduct to different comportments in the developers, thus to different results. Another external validity threat concerns the language: all the software systems are developed in JAVA. Although this alleviates parsing bias, communities using other languages could have different developer cultures and the style of emails can vary. To obtain a better generalization of the results, they should be tested on industrial systems and other object-oriented languages.

5.6 Related Work

Mining Data From Email Archives

Li *et al.* first introduced the idea of using the information stored in the mailing lists as an additional predictor for finding defects in software systems [118]. They conducted a case study on a single software system, used a number of previously known predictors and defined new mailing list predictors. Mainly such predictors counted the number of messages to different mailing lists during the development of software releases. One predictor *TechMailing*, based on number of messages to the technical mailing list during development, was evaluated to be the most highly rank correlated predictor with the number of defects, among all the predictors evaluated. Our works differs in genre and granularity of defects we predict: We concentrate on defects on small source code units that can be easily reviewed, analyzed, and improved. Also Li *et al.* did not remove the noise from the mailing lists, focusing only on source code related messages. Pattison *et al.* were the first to introduce the idea of studying software entity (function, class etc.) names in emails [154]. They used a linking based on simple name matching, and found a high correlation between the amount of discussions about entities and the number of changes in the source code. However, Pattison *et al.* did not validate the quality of their links between emails and source code. To our knowledge, our work [13] was the first to measure the effectiveness of linking techniques for emails and source code. This research is the first work that uses information from development mailing lists at class granularity to predict and to find correlation with source code defects. Other works also analyzed development mailing lists but extracting a different kind of information: social structures [36], developers' participation [135] and inter-projects migration [35], and emotional content [166].

Defect Prediction

The main difference between our work and the following approaches is that our approach is the first one to exploit email archives data for defect prediction.

Change Log-based Defect Prediction Approaches. These techniques use information extracted from the versioning system to perform defect prediction. Nagappan and Ball performed a study on the influence of code churn (*i.e.*, the amount of change to the sytem) on the defect density in Windows Server 2003 [141]. They found that relative code churn was a better predictor than absolute churn. Hassan introduced a measure of the complexity of code changes [93] and used it as defect predictor on 6 open-source systems. Moser *et al.* used a set of change metrics to predict the presence/absence of bugs in files of Eclipse [137]. Ostrand *et al.* predict faults on two industrial systems, using change and previous defect data [152]. The approach by Bernstein *et al.* uses bug and change information in non-linear prediction models [27].

Single-version Defect Prediction Approaches. These methods employ the heuristic that the current design and behavior of the program influence the presence of future defects, assuming that changing a part of the program that is hard to understand is inherently more risky than changing a part with a simpler design. Basili *et al.* used the CK metrics on 8 medium-sized information management systems [21]. Ohlsson *et al.* used several graph metrics including the McCabe cyclomatic complexity on a telecom system [147]. El Emam *et al.* used the CK metrics in conjunction with Briand's coupling metrics [40] to predict faults on one commercial Java system [66]. Subramanyam *et al.* used the CK metrics on a commercial C++/Java case study [183], while Gyimothy *et al.* performed a similar analysis on Mozilla [88]. Nagappan and Ball estimated the pre-release defect density of Windows Server 2003 with a static analysis tool [140]. Nagappan *et al.* used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems, and found that it was possible to build predictors for one individual project, but that no predictor would perform well on all the projects [142]. Zimmermann *et al.* applied a number of code metrics on the Eclipse IDE [212].

Other Approaches. Ostrand *et al.* conducted a series of studies on the whole history of different systems in order to analyze how the characteristics of source code files can predict defects [150; 151; 153]. On this basis, they proposed an effective and automatable predictive model based on such characteristics (*e.g.*, age, lines of code) [153]. Zimmermann and Nagappan used dependencies between binaries to predict defect [210]. Marcus *et al.* used a cohesion measurement based on LSI for defect prediction on C++ systems [127]. Neuhaus *et al.* used a variety of features of Mozilla to detect vulnerabilities, a subset of bugs with security risks [144]. Wolf *et al.* analyzed the network of communications between developers (*i.e.*, interactions) to understand how they are related to issues in integration of modules of a system [203]. They conceptualized communication as based on developer's comments on work items. Finally, Sarma *et al.* proposed a tool to visually explore relationships between developers, issue reports, communication (based on email archives and comments and activity on issue reports), and source code [171].

5.7 Summary

We have presented a novel approach to correlate popularity of source code artifacts within email archives to software defects. We also investigated whether such metrics could be used to predict post-release defects. We showed that, while there is a significant correlation, popularity metrics by themselves do not outperform source code and change metrics in terms of prediction power. However, we provided evidence that, in conjunction with source code and change metrics, popularity metrics can be used to increase explanative and predictive power of existing defect prediction techniques.

Reflection. Popularity metrics are merely quantitative: We do not know *why* counting the number of emails referring to a code entity is a reasonable predictor for software defects. We could guess that entities are popular and developers are talking about them because they are incorrectly designed or present other kinds of issues. To understand why our method works and further improve it, the content of each linked email should be manually inspected by conducting a qualitative investigation. This is partially done in the work we present in the next chapter and is the motivation for work we present in Part III.

Chapter 6

Supporting Program Comprehension With Emails

In Chapter 4 we presented lightweight techniques for reconnecting emails and code artifacts, and in Chapter 5 we showed a first scenario in which this novel form of information can be used to support software development: defect prediction. In this chapter, we present another scenario in which email data proved to be useful: program comprehension.

By using our lightweight traceability techniques, which provide results in a few seconds even when linking one code entity to thousands of emails, we implemented Remail, an Eclipse plugin, to make email data available in the place where developers spend most of their time–IDEs. Subsequently, we used Remail to verify that having email data at disposal in the development environment enhances tasks related to program comprehension and software development.

6.1 Overview

Developers spend most of their programming time on software maintenance and program comprehension. Between 85% and 90% of the global cost of a software system is due to software maintenance activities [67; 174], which are in a largely (up to 60%) program comprehension tasks [49].

Clear, comprehensive, and updated software documentation would be an effective approach to reduce time spent in program comprehension. Nevertheless, industrial developers report how documentation is commonly inadequate, outdated, and hard to retrieve or link to actual source code entities [114]. Open source development projects are similarly affected by issues related to documentation [82].

In small co-located development teams, unplanned face-to-face meetings are the favorite form of communication when developers face program comprehension problems [114]. Developers who need to understand source code entities (*e.g.*, to know the design rationale behind a certain implementation –the most common information need for a developer [109]), and cannot find the appropriate documentation, simply query other programmers. This solution, besides disrupting developers' attention and retaining knowledge by a few developers, is inapplicable to large or distributed development projects.

Developers, thus, replace face-to-face meetings with electronic communication means. Instant messaging, wikis, forums are viable options, but the decisive role is played by emails [73]. Considering the breadth of the information that is supposed to be found in mailing lists (*e.g.*, information about how to perform a specific development task, clarification on certain implementation details, explanation of high-level design decisions), and that readers are able to always verify the *context* of ongoing discussions and decide whether the retrieved information applies to their situation, we argue that emails can be used to help program comprehension tasks.

We present REmail, a plugin for Eclipse, which integrates email archives in the IDE. REmail is a recommendation system for emails: It allows developers to easily retrieve discussions related to the chosen code entities. The practitioner can read and learn from previous discussions occurred among programmers, thus accessing an updated and effective form of complementary documentation. Using Remail the interaction with the emails is within the development environment, *i.e.*, the programmer is not forced to frequent context switches and can retain the current development situation.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We create REmail, an Eclipse plugin based on our email-to-code linking approach.* REmail makes email data available in the IDE, *i.e.*, the place where developers spend most of their time.
- *We show that having email data at disposal enhances a number of program comprehension tasks.* We use REmail and the connection between emails and source code to complete this task.

Structure of the chapter. In Section 6.2 we present REmail, our recommender for emails and, in Section 6.3, we illustrate how we can exploit email data using it for augmenting program comprehension. In Section 6.4 we present the related work, and we summarize our contributions in Section 6.5.

6.2 REmail: Recommending Emails

When introducing the concept of open source projects, Fogel invites to use mailing lists "*as much as possible, and as conspicuously as possible*", since "*searching in them* [for answers to technical questions] *can produce answers*" [73]. Nevertheless, two critical issues hinder the effectiveness of mailing lists in supporting program comprehension and software evolution analysis:

1. Mailing lists store very large amounts of messages: The archive of Linux counts more than one million messages, the mailing lists of smaller but active projects count tens of thousands of emails. Finding, in such archives, the most relevant information concerning a specific code entity is a non-trivial task, and important discussions could be missed [98].
2. Development mailing lists discuss topics related to project development and are continuously read and written by developers. Developers spend most of their programming, designing, and understanding time within IDEs [114]. However, no matter how much related emails are to software development, programs that are *external* to IDEs manage them, sometimes even in a web browser. Therefore emails are completely *disconnected from the development environment*.

We devised REmail to tackle both issues. REmail recommends the emails that are related to specified code artifacts, by using the lightweight techniques we devised and thoroughly evaluated for this task [13; 18]. This reduces the amount of messages to be read by orders of magnitude, and lets practitioners focus on the emails related to their tasks. In addition, REmail is a modular plugin for the Eclipse IDE, thus among other benefits, it allows developers to (1) simultaneously inspect code and content of messages, easily (2) prompt recovered traceability links between code and emails, and (3) minimize the disruptive context switches necessary to access email data while programming.

Eclipse has been our target IDE because of its modular and pluggable structure, its significant amount of users (who can benefit from REmail, and provide feedback for further improvement), and its support for multiple languages. The current implementation of REmail is targeted to java systems, however our lightweight linking techniques have proven to be effective for a number of other languages (*e.g.*, ECMAScript, PHP) [18]; the multi-language support of Eclipse can be used to expand REmail to other languages with a minor effort.

Since REmail can be considered a recommendation system, we detail it by following the division described by Robillard *et al.* [169]: A recommender system involves: a *data-collection mechanism* to store development-process data and artifacts in a model, a *recommendation engine* to analyze the data and generate recommendation, and a *user interface* to trigger recommendations and present the results.

6.2.1 Data-collection Mechanism

Figure 6.1 shows the architecture we devised for REmail. Initially we devised a stand-alone solution, in which REmail was composed of the Eclipse plugin and simply used emails stored in the users' email client. This approach had disadvantages: For example, emails had to be files in MBox format and could not be modified to avoid concurrency issues with the email client; developers working on the same project could not share information such as rating; any computation (*e.g.*, finding the links between classes and emails) is done client-side and must be replicated by each client. Currently REmail is made of a server (left-hand side of Figure 6.1) and an Eclipse plugin (right-hand side).

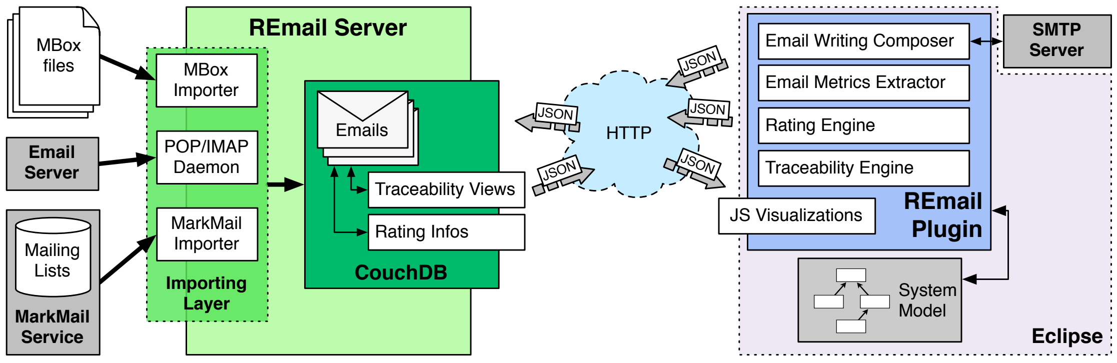


Figure 6.1: The architecture of Remail

The REmail server handles the data used by the plugin. It has to be installed by any organization willing to use REmail and it relies on the document-oriented database CouchDB[1], which provides a RESTful JSON API accessible through HTTP requests. Since emails are stored in a document-oriented database, we can change the meta-model without having to migrate data to

1 `http://couchdb.apache.org/`

a new schema, and we can use *MapReduce*[2] functions to parallelize tasks among cores or clusters of computers. Moreover, having a centralized server for storing email data has a number of advantages. For example, it allows multiple connections, thus enabling multiple developers to work simultaneously using a single email archive. This can be an advantage when rating email relevance: Whenever developers retrieve an irrelevant email (traceability links cannot be always perfect), they can rate it as non-interesting, and all the team would benefit from this information.

To populate CouchDB, we devised an `Importing Layer` that handles different sources (*e.g.*, MBox archives, MarkMail), extracts the selected emails, and instantiates them as documents in CouchDB. We also created a POP/IMAP daemon to handle newly received messages, thus keeping email data updated.

The MBox importer can also be run as a daemon to keep data from this source updated. In fact, a number of popular email clients (*e.g.*, Mozilla Thunderbird and related clients, Apple Mail, Eudora) use the MBox format to store the messages on which they operate. Having a running daemon that keeps MBox in sync is a decisive benefit: Practitioners are able to immediately take advantage of the emails already archived by their email clients, just by pointing the REmail server to the right folders. From that moment new or modified messages in the email client will be also accordingly updated in REmail.

6.2.2 Recommendation Engine

The recommendation engine analyzes the data stored in the REmail server and generates the recommendation for the provided context. Users specify the context directly from the Eclipse environment: They select the packages and classes in which they are interested and trigger the engine. The recommendation consists in presenting emails that discuss the chosen classes, as they might be useful for program comprehension and augmenting awareness.

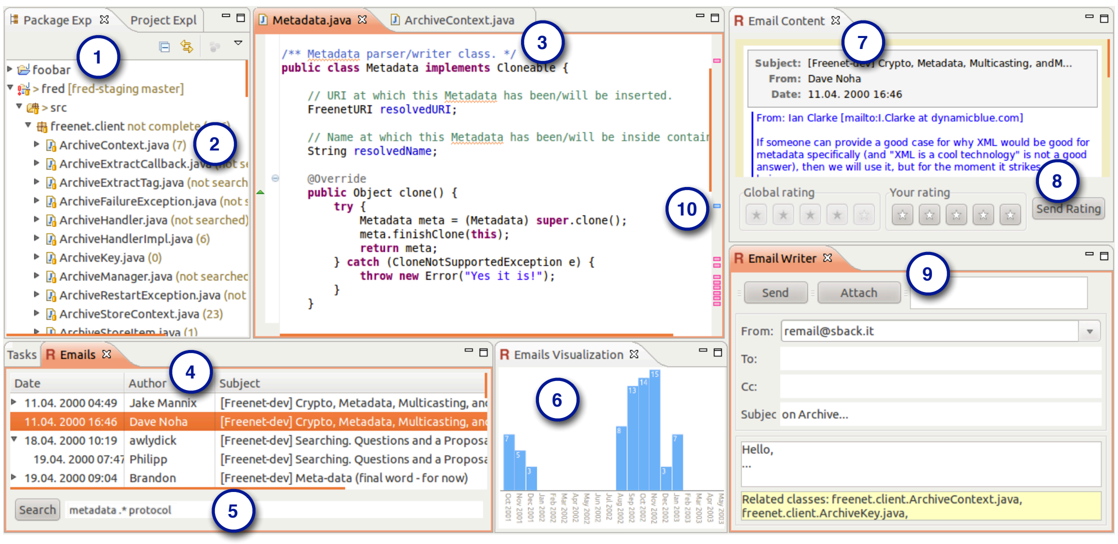
Any component of the REmail plugin that needs email data directly queries the CouchDB server through HTTP requests by sending and receiving JSON objects. As an example we see how the *Traceability Engine* works in practice. When users open a class from the *package explorer* or change tab in the main editor, REmail automatically triggers the Traceability Engine. Since the Traceability Engine retrieves the links between classes and emails by using lightweight text-matching techniques (see Chapter 4), it does not require additional information other than the fully qualified class name. The very first time users request emails for a class, REmail plugin generates a new specialized CouchDB view that implements the linking procedure, and permanently add it to the database. From that moment on, every time emails for the same class are requested, the REmail plugin will query the view to obtain the emails. The first time the view is run, CouchDB applies it to all the emails to find the appropriate documents. The results will be stored and subsequent requests will be served in real-time. Moreover, since the view results are stored in a B-tree, when new emails are stored, the view will be updated accordingly in logarithmic time. The rating engine is based on CouchDB update and view functions.

Since our lightweight methods offer different trade-offs between precision and recall, users could find some of them to be more appropriate for their needs. For this reason, we included all the methods in the Traceability Engine of the REmail server and let the server administrator configure the preferred option.

2 `http://labs.google.com/papers/mapreduce.html`

6.2.3 User Interface

Figure 6.2 presents the REMAIL plugin as seen during a development session in the Eclipse IDE.



```
foobar
> fred [fred-staging master]
  > src
    freenet.client not complete
      ArchiveContext.java (7)
      ArchiveExtractCallback.java (not se
      ArchiveExtractTag.java (not search
      ArchiveFailureException.java (not s
      ArchiveHandler.java (not searched)
      ArchiveHandlerImpl.java (6)
      ArchiveKey.java (0)
      ArchiveManager.java (not searchec
      ArchiveRestartException.java (not
      ArchiveStoreContext.java (23)
      ArchiveStoreItem.java (1)
```

```java
/** Metadata parser/writer class. */
public class Metadata implements Cloneable {

    // URI at which this Metadata has been/will be inserted.
    FreenetURI resolvedURI;

    // Name at which this Metadata has been/will be inside contai

    String resolvedName;

    @Override
    public Object clone() {
        try {
            Metadata meta = (Metadata) super.clone();
            meta.finishClone(this);
            return meta;
        } catch (CloneNotSupportedException e) {
            throw new Error("Yes it is!");
        }
    }
```

Date	Author	Subject
11.04. 2000 04:49	Jake Mannix	[Freenet-dev] Crypto, Metadata, Multicasting, anc
11.04. 2000 16:46	Dave Noha	[Freenet-dev] Crypto, Metadata, Multicasting, anc
18.04. 2000 10:19	awlydick	[Freenet-dev] Searching. Questions and a Proposa
19.04. 2000 07:47	Philipp	[Freenet-dev] Searching. Questions and a Proposa
19.04. 2000 09:04	Brandon	[Freenet-dev] Meta-data (final word - for now)

Search: metadata .* protocol

Subject: [Freenet-dev] Crypto, Metadata, Multicasting, andM...
From: Dave Noha
Date: 11.04. 2000 16:46

From: Ian Clarke [mailto:I.Clarke at dynamicblue.com]

If someone can provide a good case for why XML would be good for metadata specifically (and "XML is a cool technology" is not a good answer), then we will use it, but for the moment it strikes

From: remail@sback.it
To:
Cc:
Subjec on Archive...

Hello,
...

Related classes: freenet.client.ArchiveContext.java, freenet.client.ArchiveKey.java,

Figure 6.2: The REMAIL Plugin

Package Explorer. This is the entry point for interacting with REMAIL and triggering its recommendation engine (Figure 6.2, Point 1). The first time REMAIL is used, the developer selects the classes and packages of interest and starts the search by clicking *REMAIL search* in the popup menu. By selecting a package, the search will be recursively performed in all the subclasses. Once the process is completed, next to each chosen entity in the Package Explorer, the user sees a number (Point 2), which shows how many "hits" each entity has within the mailing list, thus effectively measuring the *popularity* of the entities (see Chapter 5). As we discuss in Section 6.3, the popularity can be used as an entry point to study an unknown system.

Emails View. Once a search has been performed, the user can click on any indexed class in the package explorer, or open/change a class in the editor (Point 3) and the *Emails View* (Figure 6.2, Point 4) will be updated accordingly. The visualization used in this panel conveys multiple details: 1. The messages are sorted by time, 2. the three columns (namely *date*, *author*, and *subject*) chunk the main metadata, and 3. the nested tree layout preserves the discussion *threads*. This view is similar to the one presented in common email clients: It scales to a vast number of emails and gives a temporal dimension. The latter feature is interesting from a program comprehension point of view: Even though the source code is updated, it is possible to walk back in time, by reading the emails discussing a class in the past. In mailing lists emails are organized in threads: Whenever there is a reply on an email, subsequent emails are handled in discussion threads by email clients. REMAIL supports threads in the *Emails View*, and in the *Email Content View*. This enhances email readability and the discussion context is more explicit. With the search box (Point 5), they can further refine their search using keyword and/or regular expressions.

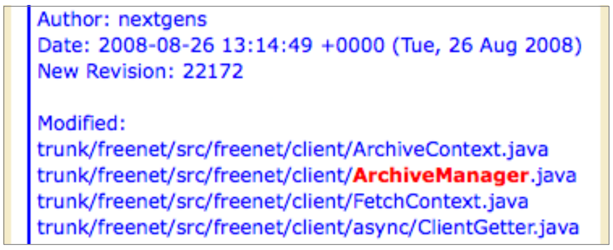
Emails Visualization. With this view (Point 6), users can see how the related emails are distributed in time. This view shows the emails related to the chosen class as a bar chart: The x-axis is a discrete timeline, split in bins of equal duration, and the height is the number of emails exchanged during each period of time. This graph allows developers to see *trends* in discussions related to the chosen class. When users find a significant period and click on a bar, the *Emails view* (Point 4) shows only those emails. This view is based on HTML and Javascript. By using the SWT Browser, one can include HTML pages and interactive javascript within Eclipse and use java-to-javascript and javascript-to-java callbacks to make the view interacting with the environment. In this way, the same visualization can be accessed through a web browser outside of Eclipse (we can imagine a manager using this feature), all the effective javascript visualization libraries that are available can be used, and the IDE is more responsive than when using Eclipse visualizations.

Email Content View. Point 7 in Figure 6.2 details the panel that is opened when a specific email is selected in the *Emails View*. Its content is also visual: A box presents the metadata, different colors and bars distinguish the different quotation levels, and a bold red typeface highlights the name of the class for which the email was recommended. Users can also rate the importance/relevance of emails in this view (Point 8), this rating will be shared and averaged with the other users, thus creating a *social* rating.

Email Writer View. With REmail, users can not only *consume* email information, but also *produce* it. By selecting any number of entities in the package explorer and/or a snippet in the code editor and clicking a button a new email is automatically prepared, whose body includes the chosen information. The *Email Writer* view (Point 9) shows the intermediate result. Users can then complete the message by hand. The code snippets and fully qualified names of entities are automatically included in the email body.

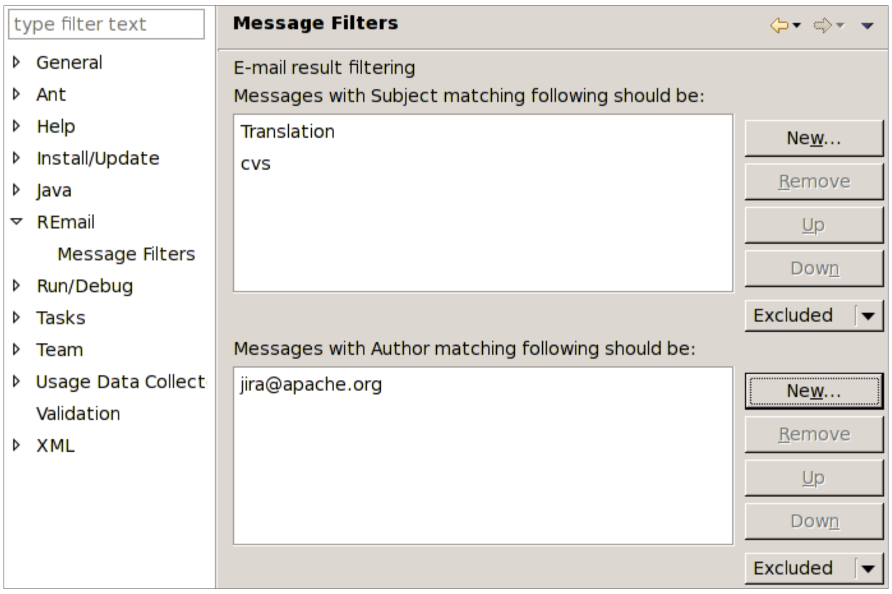
Editor support Most time that developers spend using IDEs is focused on the editor, where they actually work with the source code. They might want to maximize the editor to completely fill the screen. By doing so, the views of REmail are not visible. Therefore, we enhance the Editor itself to provide support in this situation. *Markers* (Point 10) signal general point of interest in any of the resource files. We have used the bookmark markers to provide information about all the class names visible in the source code editor, to which some emails have been linked. A toolbar button triggers markers, so that user can decide whether to show them.

Global email filtering During our case studies we obtained a considerable number of emails, that indeed referred to a class in question, but were irrelevant in a program comprehension context. The vast majority of such irrelevant messages are automatically generated and sent by issue repository systems, or by version control systems for detailing commits (*e.g.*, see Figure 6.3). These emails include listings of all classes that are part of each commit or defect report, which is hardly relevant for their understanding.


```
Author: nextgens
Date: 2008-08-26 13:14:49 +0000 (Tue, 26 Aug 2008)
New Revision: 22172

Modified:
trunk/freenet/src/freenet/client/ArchiveContext.java
trunk/freenet/src/freenet/client/ArchiveManager.java
trunk/freenet/src/freenet/client/FetchContext.java
trunk/freenet/src/freenet/client/async/ClientGetter.java
```

Figure 6.3: Excerpt of a related, however irrelevant, linked email



Message Filters

E-mail result filtering

Messages with Subject matching following should be:

Translation

cvs

Excluded

Messages with Author matching following should be:

jira@apache.org

Excluded

Figure 6.4: Email filtering configuration

REmail includes a message filtering feature to reject messages based on subject and author fields. The filtering configuration panel is shown in Figure 6.4. This feature proved to be helpful during our case study: The emails posted to the mailing lists by the version control systems have a special subject and all the emails posted by the issue repository system have the same sender, thus, by creating filters, we had been able remove all unnecessary emails. The filtered emails are removed from the linking results and do not appear in *Email View*, nor they are counted as "hits". The filtered links are not removed from the caching mechanism, thus we can try different filters without triggering the search again.

6.3 Program comprehension through emails

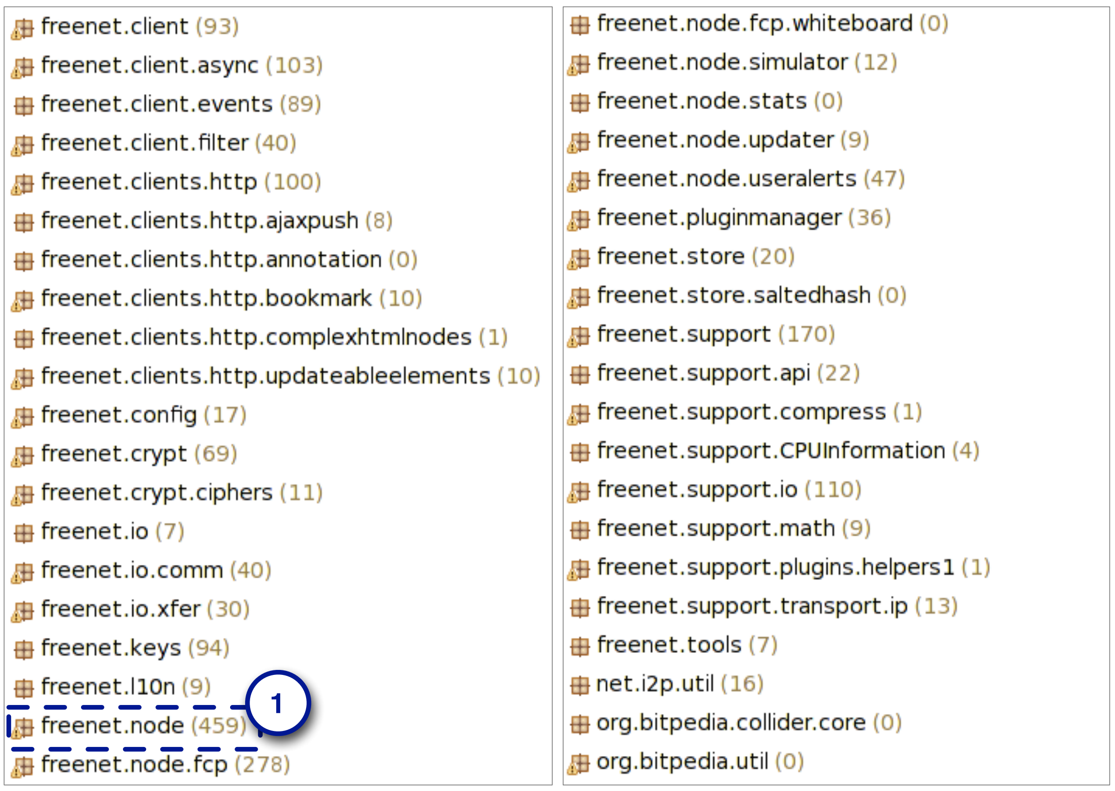
We present scenarios to illustrate the benefits of the program comprehension support that REmail offers by recommending emails. We explore two OSS systems, Apache Mina and Freenet, from unrelated domains, with different size, and with distinct communities (see Section 3.4).

6.3.1 Entry points from class popularity in emails

A key issue in program comprehension efforts is to know where to start. Email data provides both qualitative and quantitative information for this purpose. The augmented Package Explorer view, which shows decorations with the number of emails related to the chosen packages and classes, gives hints on the "popularity" of entities in the mailing list, in quantitative terms. We argue that this value might be high in classes that implement the core functionalities of a system, thus it might be used for recommending *entry points* for program comprehension.

This “secondary” recommendation, based on popularity, can be easily *contextualized* and evaluated by the practitioner, thanks to the qualitative aspect of emails. In other words, this popularity is not a value coming out of the blue, but, on the contrary, it is supported by the content of the emails: By skimming the messages’ text, one can decide whether a popular class is worth understanding for the task at hand.

Freenet



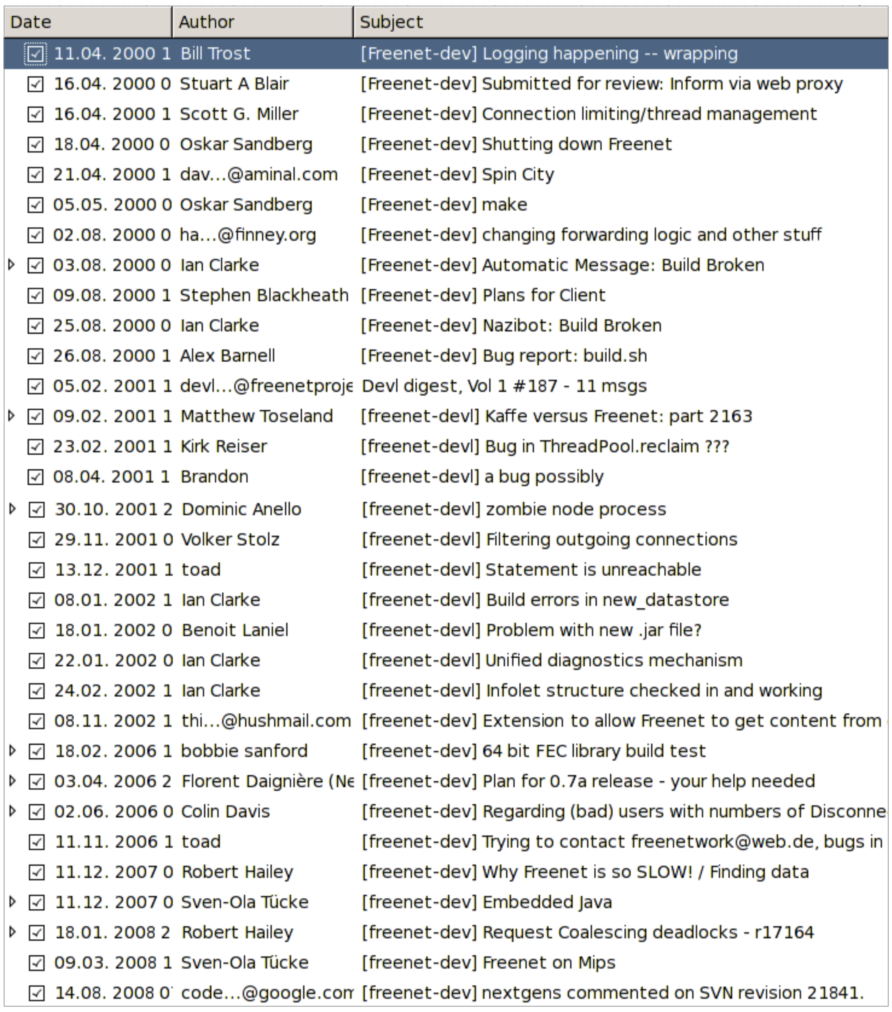
freenet.client (93)
freenet.client.async (103)
freenet.client.events (89)
freenet.client.filter (40)
freenet.clients.http (100)
freenet.clients.http.ajaxpush (8)
freenet.clients.http.annotation (0)
freenet.clients.http.bookmark (10)
freenet.clients.http.complexhtmlnodes (1)
freenet.clients.http.updateableelements (10)
freenet.config (17)
freenet.crypt (69)
freenet.crypt.ciphers (11)
freenet.io (7)
freenet.io.comm (40)
freenet.io.xfer (30)
freenet.keys (94)
freenet.l10n (9)
freenet.node (459)
freenet.node.fcp (278)
freenet.node.fcp.whiteboard (0)
freenet.node.simulator (12)
freenet.node.stats (0)
freenet.node.updater (9)
freenet.node.useralerts (47)
freenet.pluginmanager (36)
freenet.store (20)
freenet.store.saltedhash (0)
freenet.support (170)
freenet.support.api (22)
freenet.support.compress (1)
freenet.support.CPUInformation (4)
freenet.support.io (110)
freenet.support.math (9)
freenet.support.plugins.helpers1 (1)
freenet.support.transport.ip (13)
freenet.tools (7)
net.i2p.util (16)
org.bitpedia.collider.core (0)
org.bitpedia.util (0)

Figure 6.5: Excerpt from Package Explorer: Freenet packages with popularity

Figure 6.5 reports the popularity of Freenet packages, as shown by REmail’s Package Explorer decorator. It shows that the most popular package is `freenet.node` (Point 1), with classes discussed in more than 450 emails. The second most discussed package is `freenet.node.fcp`, with slightly more than 250 emails, while the other packages are significantly less popular.

We investigate the most popular package: In the `node` package, developers mainly discuss four entities: classes `Node` (74 emails), `PacketSender` (61), and `PeerNode` (67), and interface `Request-Client` (98). With a brief analysis of the emails for the interface, we see that it was popular during the first phases of Freenet development, but afterwards its importance gradually faded. We focus on the other classes that are still currently discussed:

Node: By looking at the distribution of the emails over time, via the *Emails View* panel, we see (Figure 6.6) that the large class `Node` has been very popular since the inception of the project (*i.e.*, year 2000). By inspecting the code, we discover not only that it includes the main method from which the Freenet system bootstraps, but also that it models the node run by the user in the network. Since Freenet is a peer-to-peer system, the user node has a crucial role for the whole application and is essential for comprehending how the software functions as a whole.

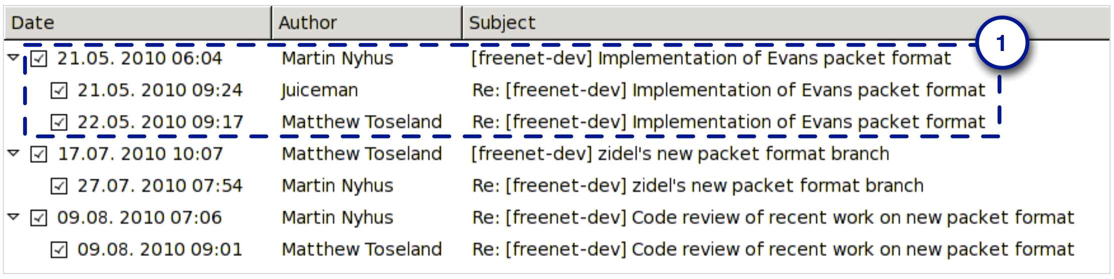


Date	Author	Subject
☑ 11.04. 2000 1	Bill Trost	[Freenet-dev] Logging happening -- wrapping
☑ 16.04. 2000 0	Stuart A Blair	[Freenet-dev] Submitted for review: Inform via web proxy
☑ 16.04. 2000 1	Scott G. Miller	[Freenet-dev] Connection limiting/thread management
☑ 18.04. 2000 0	Oskar Sandberg	[Freenet-dev] Shutting down Freenet
☑ 21.04. 2000 1	dav...@aminal.com	[Freenet-dev] Spin City
☑ 05.05. 2000 0	Oskar Sandberg	[Freenet-dev] make
☑ 02.08. 2000 0	ha...@finney.org	[Freenet-dev] changing forwarding logic and other stuff
▷ ☑ 03.08. 2000 0	Ian Clarke	[Freenet-dev] Automatic Message: Build Broken
☑ 09.08. 2000 1	Stephen Blackheath	[Freenet-dev] Plans for Client
☑ 25.08. 2000 0	Ian Clarke	[Freenet-dev] Nazibot: Build Broken
☑ 26.08. 2000 1	Alex Barnell	[Freenet-dev] Bug report: build.sh
☑ 05.02. 2001 1	devl...@freenetproje	Devl digest, Vol 1 #187 - 11 msgs
▷ ☑ 09.02. 2001 1	Matthew Toseland	[freenet-devl] Kaffe versus Freenet: part 2163
☑ 23.02. 2001 1	Kirk Reiser	[freenet-devl] Bug in ThreadPool.reclaim ???
☑ 08.04. 2001 1	Brandon	[freenet-devl] a bug possibly
▷ ☑ 30.10. 2001 2	Dominic Anello	[freenet-devl] zombie node process
☑ 29.11. 2001 0	Volker Stolz	[freenet-devl] Filtering outgoing connections
☑ 13.12. 2001 1	toad	[freenet-devl] Statement is unreachable
☑ 08.01. 2002 1	Ian Clarke	[freenet-devl] Build errors in new_datastore
☑ 18.01. 2002 0	Benoit Laniel	[freenet-devl] Problem with new .jar file?
☑ 22.01. 2002 0	Ian Clarke	[freenet-devl] Unified diagnostics mechanism
☑ 24.02. 2002 1	Ian Clarke	[freenet-devl] Infolet structure checked in and working
☑ 08.11. 2002 1	thi...@hushmail.com	[freenet-dev] Extension to allow Freenet to get content from
▷ ☑ 18.02. 2006 1	bobbie sanford	[freenet-dev] 64 bit FEC library build test
▷ ☑ 03.04. 2006 2	Florent Daignière (Ne	[freenet-dev] Plan for 0.7a release - your help needed
▷ ☑ 02.06. 2006 0	Colin Davis	[freenet-dev] Regarding (bad) users with numbers of Disconne
☑ 11.11. 2006 1	toad	[freenet-dev] Trying to contact freenetwork@web.de, bugs in
☑ 11.12. 2007 0	Robert Hailey	[freenet-dev] Why Freenet is so SLOW! / Finding data
▷ ☑ 11.12. 2007 0	Sven-Ola Tücke	[freenet-dev] Embedded Java
▷ ☑ 18.01. 2008 2	Robert Hailey	[freenet-dev] Request Coalescing deadlocks - r17164
☑ 09.03. 2008 1	Sven-Ola Tücke	[freenet-dev] Freenet on Mips
☑ 14.08. 2008 0	code...@google.con	[freenet-dev] nextgens commented on SVN revision 21841.

Figure 6.6: Excerpt from Emails View: emails recommended for class `Node`

PacketSender: This class implements packet sending through the Freenet network. It has a general importance in the system, and by reading one recommended email, we understand that it is *critical* for a developer who must deal with message handling: "*Are you interested in implementing message priorities? MessageItem and PacketSender are the most relevant classes.*" This message also reveals a hidden coupling, not detectable by static analysis, with `MessageItem`.

PeerNode: The opening code comment of `PeerNode` states that it "*represents a peer we are connected to.*" Therefore, it plays a central role in the Freenet functioning and is another important entry point for program comprehension. Moreover, by reading among the most recent email threads recommended by REmail (Figure 6.7, Point 1), we discover additional information that could not have been learned by solely investigating the code. `PeerNode` is responsible for implementing the Freenet Network Protocol (FNP) –the communication protocol used in Freenet. A developer who must change this protocol is required to "*move all the FNP related code from PeerNode to a new class, and have PeerNode use the old code through this class. The new code can then be added without touching FNP, and PeerNode* [can] *choose which format to use for each peer.*" By reading the same thread, the developer interested in modifying FNP would also discover the other two classes responsible for the implementation of FNP: `PacketTracker` and `SessionKey`.



Date	Author	Subject
☑ 21.05. 2010 06:04	Martin Nyhus	[freenet-dev] Implementation of Evans packet format
☑ 21.05. 2010 09:24	Juiceman	Re: [freenet-dev] Implementation of Evans packet format
☑ 22.05. 2010 09:17	Matthew Toseland	Re: [freenet-dev] Implementation of Evans packet format
☑ 17.07. 2010 10:07	Matthew Toseland	[freenet-dev] zidel's new packet format branch
☑ 27.07. 2010 07:54	Martin Nyhus	Re: [freenet-dev] zidel's new packet format branch
☑ 09.08. 2010 07:06	Martin Nyhus	Re: [freenet-dev] Code review of recent work on new packet format
☑ 09.08. 2010 09:01	Matthew Toseland	Re: [freenet-dev] Code review of recent work on new packet format

Figure 6.7: Emails View: recent threads recommended for `PeerNode`

To further evaluate the importance of these three classes in the system, we analyzed them in terms of *Design Flaws* [113]. The detection strategies we use (see [128]) diagnose all the three classes as affected by the *Behavioral God Class* [164] design flaw, *i.e.*, they tend to incorporate a disproportionately large amount of intelligence. This reinforces the hypothesis that they represent an important entry point for program comprehension. Additionally, all the classes present other design flaws such as *Brain Methods*, *Intensive Coupling*, and *Shotgun Surgery*. We do not analyze each design flaw, but refer the interested reader to [113]

Mina

Mina is an application framework for supporting the development of network applications. Within the project documentation, developers offer a decomposition of the system to introduce newcomers to its architecture. Figure 6.8 replicates this decomposition.

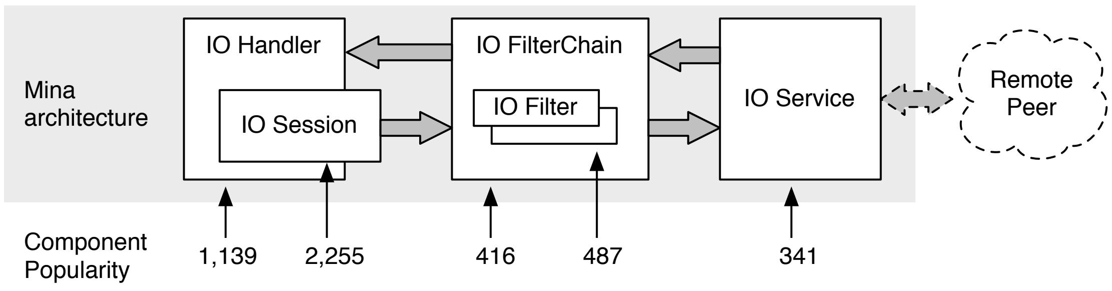


Figure 6.8: Mina architecture: Main components and their popularity

With this decomposition in place, we compare it to the popularity of its component. For each component (*e.g.*, *IO Handler*), we report the popularity of the corresponding class (*e.g.*, the class `core.service.IOHandler`). Only in the case of *IO Filter Chain*, we sum the popularity of the interface `IOFilterChain` and of its sole implementer `DefaultIOFilterChain`.

The popularity values depicted in Figure 6.8 show that the most discussed components match the architecture proposed by the developers. In fact, only a few other classes reach such a popularity score: `IOBuffer` (541 emails), the replacement of the Java library class `ByteBuffer`, necessary for writing on any *IO Session*; `IOAcceptor` (390 emails) and `IOConnector` (347) used for starting a server and a client, respectively; and `ProtocolCodecFilter` (1,010), a specialized IO filter also detailed in the official decomposition. In other words, we could have extracted an almost equivalent decomposition simply by using quantitative data from emails.

On an unrelated note, we see how the number of emails talking about the most popular classes is rather higher compared to Freenet, even though the respective mailing lists archive a similar total number of messages. This reflects the different programming and communication habits between the two development communities.

6.3.2 Software Evolution Analysis

Version control (or software configuration management–SCM) systems offer historical information on the evolution of the source code. They can be used to track *changes* in source code artifact in order to detect whether a class is stable or always morphing. For example, researchers have successfully used *change metrics* to predict defects [27; 137].

We argue that, in system in which developers mainly communicate in mailing lists (such as open source projects, and distributed teams), emails might be used to *complement* known information in order to better understand the relevance of changes. The rationale is that classes that are not discussed in the development mailing lists are likely to be more stable and less prone to major modifications, since a substantial change would require an agreement of the team.

To investigate our hypothesis, we analyze the package `util` in the Mina project. It presents 98 distinct commits in fifty months, and discussions in the mailing list. It contains 17 classes, of which we manually inspected the complete history in the SCM system and all the emails recommended. We analyze the changes occurred since the inception of the mailing list (2006).

Figure 6.9 visualizes the results of our analysis. We divide the time in one-month slots and fill them with the occurred events. Grey boxes represent related emails, the other boxes represent commits. Light blue boxes represent class addition, white boxes represent minimal changes (*e.g.*, author renaming, license change, reformatting), and red boxes represent relevant changes that modified a class' behavior.

The first thing we note is the small number of relevant changes: Only 6 relevant changes over 98 commits. From our hypothesis, we expected this behavior, since the mailing list is silent on most of the classes. We note how there is no relevant change in classes not mentioned in the mailing list. Concerning the relevant changes, we see (circled in Figure 6.9) how these happen close in time to discussion related to the class. The only exception is `IdentityHashSet`, where the related discussion happens after many months.

An analogous pattern is present in the Freenet system. For example, the package `crypt` contains classes that used to be discussed in the past of the project, but that have been almost no discussed for two years. By analyzing the related commits, we verify that they contain no relevant change. From this scenario, we see how historical information provided by emails might help in understanding where the current, active, and relevant development is focused.

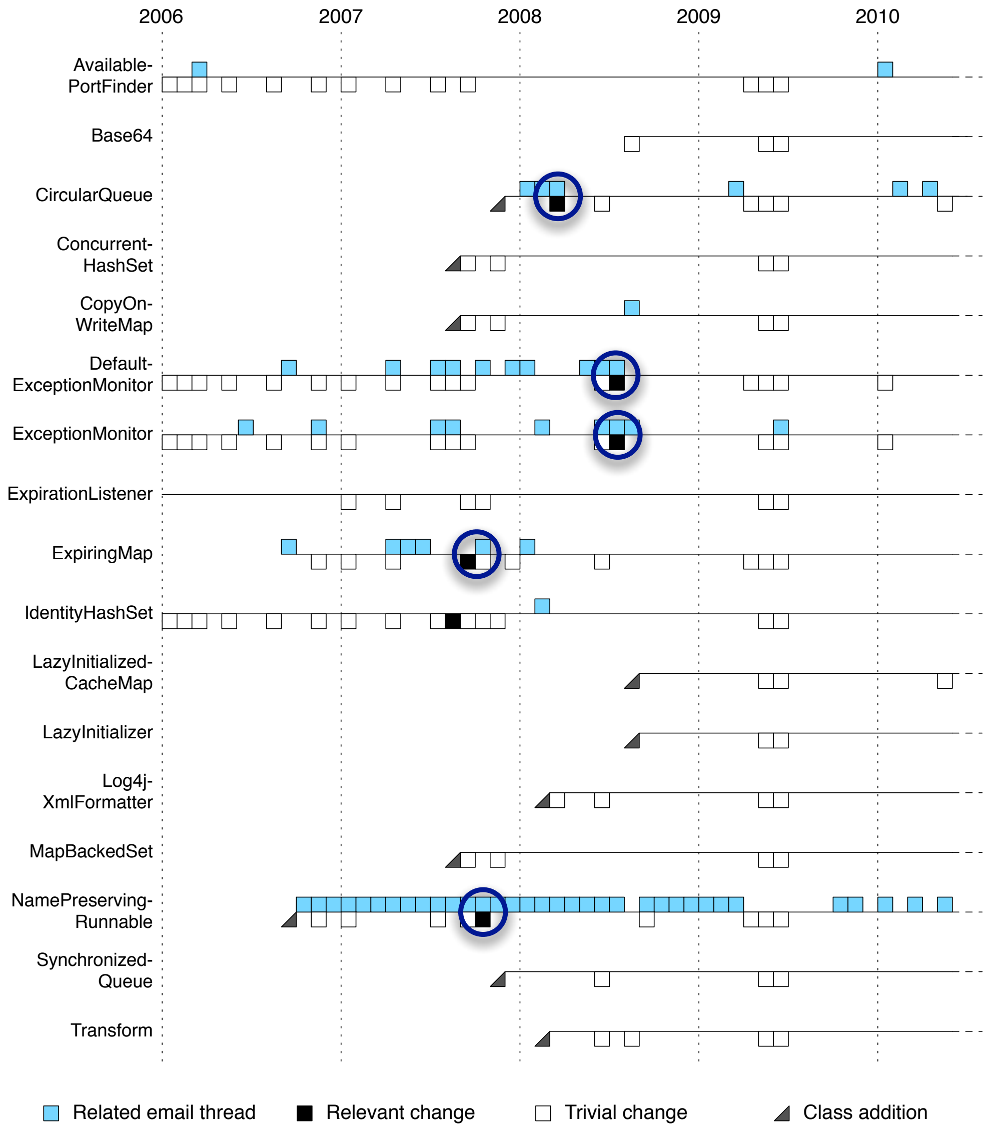


Figure 6.9: Package `org.apache.mina.util`: changes and discussions

6.3.3 Expert finding

Program comprehension involves keeping up with who on a distributed team is expert about specific code entities. Given the complexity and the amount of changes in software, this is a non-trivial task. Researchers have proposed a number of approaches to recommend experts (*e.g.*, [189; 135; 173]), most of them based on authorship of code: The person committing changes to an artifact has expertise in it. Emails recommended by REMAIL also report the author (see Point 4 and 7 of Figure 6.2). We argue that this information can be used to extract both quantitative and qualitative information about the expertise on discussed entities. As an example, we see how REmail can be used to find an expert of class`BookmarkItem`.

We first selected the entity itself (see Figure 6.2, Point 1), which has 26 related emails. Then, in the *Emails View* (Point 4), we already see how Toseland wrote several emails on this entity. This quantitative information suggests us his potential class expertise. To confirm this belief, we select one of the emails he wrote and we read it in *Email Content View* (Point 7). Thanks to the different colors used to distinguish the quotation levels, we clearly read that Toseland, first, indicated how `BookmarkItem` must be aggregated: "*you should put each BookmarkItem as a subfieldset, not a string*"; then, gave the rationale behind this behavior: "[to have] *the ability to easily add fields.*" Using REmail we found an expert of this entity.

Table 6.1: Code commits involving `org.apache.mina.util.ExceptionMonitor`

Revision	Date	Author	Revision	Date	Author
995776	Oct 2010	elecharny	671827	Jun 2008	jvermillard
900040	Jan 2010	elecharny	576217	Sep 2007	trustin
783334	Jun 2009	elecharny	565669	Aug 2007	trustin
774593	May 2009	elecharny	555855	Jul 2007	trustin
678335	Jul 2008	mwebb	497314	Jan 2007	trustin

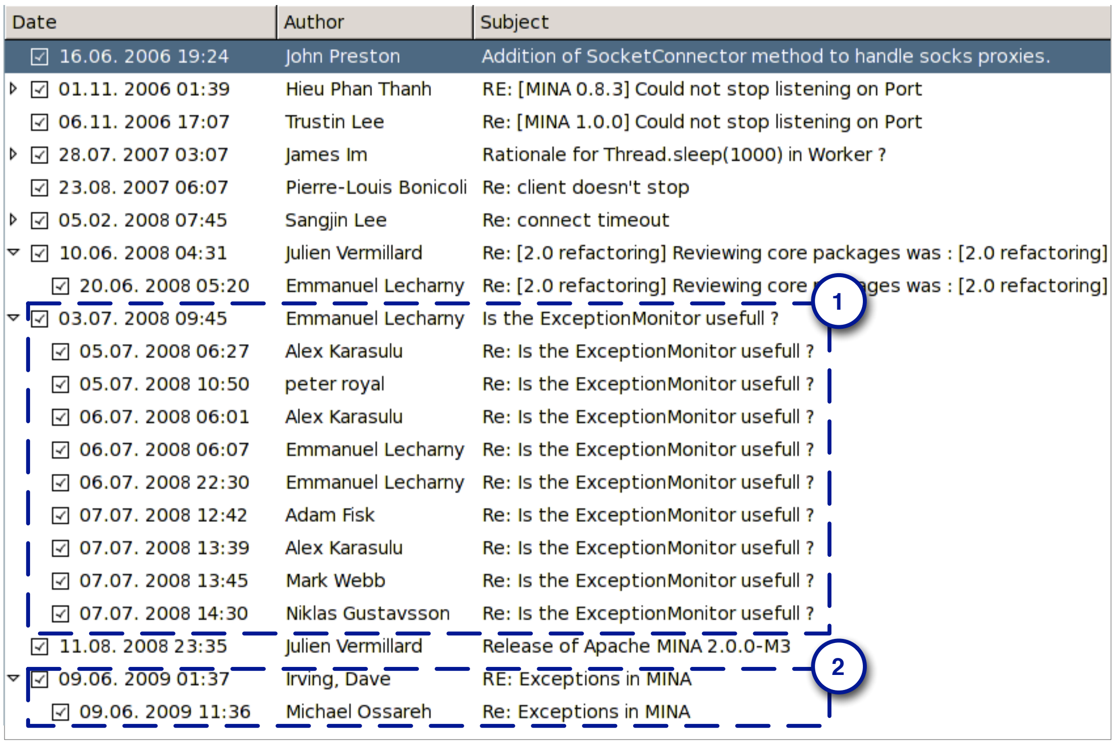
As another example, we see how REmail *complements* the expertise information we find in the SCM system. We analyze the Mina class `ExceptionMonitor`. Table 6.1 reports the SCM system commits involving this class, by date and author. Only four developers committed on this class: They are the exclusive experts from a SCM system point of view.

Figure 6.10 reports the emails related to `ExceptionMonitor`, as recommended by REmail. The aforementioned committers were all involved in discussions, in a moment in time close to their activity on the class. For example, we see that Trustin Lee (trustin, in commits) wrote emails in 2006 and worked on the class up to 2007. In 2008, Mark Webb and Julien Vermillard where both committing and discussing the class. Currently Emmanuel Lecharny is the sole committer.

By reading the thread in Point 1, we learn that other developers designed the class, but are now no longer involved in the implementation (*e.g.*, Karasulu: "*I think this was something Trustin and I talked about while experimenting with Monitors versus logging.* [This class] *was a bad idea then and I think it is a bad idea now.*"). At the same time, by reading Point 2, we see that Dave Irving is knowledgeable about the functioning of the class: "*an ExceptionMonitor instance* [...] *doesn't swallow exceptions (or captures them for relaying back to your test if you're not running off the main thread).*" Only with the SCM data, we would have not been able to discover his expertise.

6.3.4 Recovering Additional Information

Official documentation, *e.g.*, design documents, is regarded by developers as "write-only media," as it is difficult and cumbersome to evolve along with the rest of the systems [114]. Moreover, not all the developers have the rights to modify it. On the contrary, emails are easier to write, due to their less formal nature, and can be written by non-developers. We claim that recommended emails might contain *complementary* information not available in code comments, or official documentation, thus helping program comprehension.



Date	Author	Subject
16.06. 2006 19:24	John Preston	Addition of SocketConnector method to handle socks proxies.
01.11. 2006 01:39	Hieu Phan Thanh	RE: [MINA 0.8.3] Could not stop listening on Port
06.11. 2006 17:07	Trustin Lee	Re: [MINA 1.0.0] Could not stop listening on Port
28.07. 2007 03:07	James Im	Rationale for Thread.sleep(1000) in Worker ?
23.08. 2007 06:07	Pierre-Louis Bonicoli	Re: client doesn't stop
05.02. 2008 07:45	Sangjin Lee	Re: connect timeout
10.06. 2008 04:31	Julien Vermillard	Re: [2.0 refactoring] Reviewing core packages was : [2.0 refactoring]
20.06. 2008 05:20	Emmanuel Lecharny	Re: [2.0 refactoring] Reviewing core [illegible]ges was : [2.0 refactoring]
03.07. 2008 09:45	Emmanuel Lecharny	Is the ExceptionMonitor usefull ?
05.07. 2008 06:27	Alex Karasulu	Re: Is the ExceptionMonitor usefull ?
05.07. 2008 10:50	peter royal	Re: Is the ExceptionMonitor usefull ?
06.07. 2008 06:01	Alex Karasulu	Re: Is the ExceptionMonitor usefull ?
06.07. 2008 06:07	Emmanuel Lecharny	Re: Is the ExceptionMonitor usefull ?
06.07. 2008 22:30	Emmanuel Lecharny	Re: Is the ExceptionMonitor usefull ?
07.07. 2008 12:42	Adam Fisk	Re: Is the ExceptionMonitor usefull ?
07.07. 2008 13:39	Alex Karasulu	Re: Is the ExceptionMonitor usefull ?
07.07. 2008 13:45	Mark Webb	Re: Is the ExceptionMonitor usefull ?
07.07. 2008 14:30	Niklas Gustavsson	Re: Is the ExceptionMonitor usefull ?
11.08. 2008 23:35	Julien Vermillard	Release of Apache MINA 2.0.0-M3
09.06. 2009 01:37	Irving, Dave	RE: Exceptions in MINA
09.06. 2009 11:36	Michael Ossareh	Re: Exceptions in MINA

Figure 6.10: Emails recommended for `mina.util.ExceptionMonitor`

We present the example of the class `CircularQueue`, in the MINA system, which provides anecdotal evidence of our hypothesis. The SCM system (as shown in Table 6.1) reports five commits on this class in the last months, thus underlying a sort of evolution. However, from a previous scenario we know that these changes are not relevant and the class is almost frozen. For this reason, one would wonder its real purpose, which components use this class, and why it is stable. The only information provided by the class comments and documentation is that `CircularQueue` implements an "unbounded circular queue based on array." However, we can read a recent thread titled "About CircularQueue", among those recommended by REMAIL as pertaining to `CircularQueue`, to gain additional information. From the emails we discover that this class has a logical connection to *ConcurrentLinkedQueue*. Even though the latter "*performs bad comparing to synchronized* [CircularQueue] *when the number of accessing threads are very small*", developers decided to remove "*all references to the non-thread-safe* [CircularQueue] *data structure, and replace it with a* [ConcurrentLinkedQueue]." The reason is that "*not only* [ConcurrentLinkedQueue] *is a comparable data structure, but it's also thread safe, and tested.*" Eventually, we read that developers are considering to "*remove the* [CircularQueue] *data structure from the code base*" as it should only be used by the core and not by framework users.

Thanks to this additional information, we learned what the most important issues of the class are (*i.e.*, not-thread-safe and not well tested), which components use it (*i.e.*, only internal core classes), and why it has not been changed significantly lately (*i.e.*, it will probably be removed).

6.4 Related Work

Holmes and Begel devised an approach similar to ours, including external artifacts in the development environment [97]. They presented *Deep Intellisense*, a Visual Studio IDE plugin that links a number of artifacts (*e.g.*, bug reports, e-mails, code changes) to source code entities. It features three views: a structural view of the artifact chosen, a view to represent people related to that artifact, and a view to display all the related historical information (*e.g.*, checkins, or bug reports). As opposed to our approach, Deep Intellisense always relies on external applications to handle the visualization of different artifacts: For example, when a user clicks on a bug, the native viewer is open. By implementing Remail, we strive to give the users views that are consistent with the development environment, in order to avoid unnecessary context switches, and to allow more interaction. For example, as our future work, we plan to allow the developer to click on the content of an email to trigger appropriate events in the IDE. This would not be possible by relying on external viewers. In addition, Deep Intellisense needs a specialized database with all the information modeled, in order to use its implicit query system [200]. By using MBox files, we removed the burden of such complex infrastructure from users' shoulders, thus lowering the bar for adopting Remail in everyday development.

Hipkat offers an integrated approach to access information stored in project archives [199]. Similarly to our approach, it is an Eclipse plugin. The main difference resides on fact that Hipkat requires an external server process to monitor sources, to store them in a database, to identify links, and to reply to requests sent by the client. While, in Remail, all these operations are conducted in the client, since we strived to implement our approach as fast, lightweight, and unobtrusive. In addition, the users of Remail do not need to specify any query for retrieving the relevant data, they simply choose the entities, and emails are automatically recommended.

IBM Jazz[3] offers a framework, built on top of Eclipse, to support collaborative software development. It features IM communication in the IDE, and uses the concept of "work items" to track and coordinate development tasks and workflows. Each work item is connected to other artifacts (*e.g.*, builds, defect reports, change sets, or source code entities). Such work items have analogies with emails, even though the latter might have a broader perspective. While Jazz advises the use of a brand new technology, we decided on harnessing the power of emails, a pre-existing popular communication means successfully adopted by developers of a vast majority of software projects. This allows developers to take advantage of mailing lists that archive years of relevant information about the development of software systems.

Mylyn offers the concept of "task context" that focuses on automatically link all relevant artifacts to the task-at-hand [104]. As for Remail, this system helps in reducing information overload and easing the sharing of expertise. However, as for the case of Jazz, it also requires an additional technology to be used, thus it does not provide access to differently archived data and requires a higher learning effort by developers. At the same time, Remail and Mylyn can coexist in the same IDE and provide complementary data.

6.5 Summary

We presented a new approach for program comprehension based on email information. We implemented REmail, an Eclipse plugin to integrate email communication. It enables the con-

3 `http://jazz.net/`

nection between code artifacts and emails, within the programming environment. By using code-to-emails lightweight linking techniques, REmail allows the user to easily retrieve discussion relevant to the chosen entities. REmail revolves around two aspects:

1. *Simplicity:* REmail allows developers to take advantage of email archives already present in their common email clients and to easily import new archives via MarkMail or Mailman. Also, it lets practitioners find and read relevant emails, for a chosen entity, with one click.
2. *Integration:* REmail smoothly integrates with email clients: When REmail uses an archive of a client, it will not interfere with its functioning, but, on the contrary, will take advantage of the updates performed by the user from the client, by updating its own data. In addition, new archives imported by REmail can immediately also be used by email clients. Also, REmail integrates with the IDE: It has internal views to avoid context switches and ease concurrent code and email inspection.

We have shown how the email information, as displayed by REmail, helps to find entry points in an unknown system, understand software evolution, identify experts, and complement missing documentation. The main strength of REmail resides in the fact that it recommends emails–discussions in a context.

Limitations. The main limitation of the study presented in this chapter is that it was validated only on two OSS systems, and that the evidence is so far anecdotal. For example, concerning the entry point analysis (see Section 6.3.1), in the studied systems we found that classes are popular because they form the system core. Nevertheless, artifacts in other systems might be popular in emails for other reasons (*e.g.*, because of their size, or because they are subject to frequent changes); it would be necessary to analyze whether these classes can also be considered as valid entry points for comprehending a system.

Reflection. The main contribution of REmail is disclosing both *qualitative* and *quantitative* information provided by email archives, so that it can be used during software development to support program comprehension. Remail recommends emails, which are discussions in a context. The context is vital for users to verify the value of a recommendation. While with popularity metrics, we only provide a mere number, which does not provide further insights about developers' thoughts or recommendations, Remail recommendations provide the qualitative aspect of emails' content.

While analyzing the emails for our case study, we realized how the content of email is extremely noisy and composed of different languages: We find not only natural language sentences, but also entire parts written in different programming languages in forms of code snippets, patches, or execution traces. This situation impacts different aspect of mining development emails. For example, we noticed that artifacts mentioned in natural language parts of emails are more relevant to program comprehension, than artifacts mentioned in other contexts (*e.g.*, stack traces). Moreover, the emails' noise *must* be removed to provide meaningful data for further research [32]. With this study we discovered the second challenge in proving our thesis (see Section 1.2): Dealing with the noisy and mixed language content of development emails.

Part III

Structuring Unstructured Software Data

Obtaining benefits from software repositories, to support program comprehension and software development, is not trivial: The information extracted from software repositories must be relevant, unbiased, *and its contribution* comprehensible. *In this vein, researchers are analyzing the quality of issue datasets, for example to verify if approaches are accurate enough to provide flawless information [34; 187], and to determine what information is more relevant (*e.g.*, in issue reports [211] or among the changes in version histories [102]).*

*When we extract information from repositories containing natural language documents (*e.g.*, web sites, IRC chat logs, mailing lists), we have to be careful about the quality of the data. In fact, natural language leaves complete freedom to document authors, not enforcing the rules on data input that we find in structured data (*e.g.*, source code, issue trackers), which is to be produced or parsed by a machine. On this note, Bettenburg* et al. *presented the risks of using email data without a proper cleaning pre-processing phase [32], with compelling "before and after" examples to show how noise severely impacts such data.*

Even when an accurate data cleaning phase is conducted, documents are mostly treated as bags of words*: a count of which terms appear and how frequently. This simplification is proven to be effective in the information retrieval field, where techniques are tested on well-formed natural language documents, generated by information professionals, such as journalists, lawyers, and doctors [123]. In software engineering, although effective for some tasks (*e.g.*, traceability between documents and code [3]), this method reduces the quality, reliability, and comprehensibility of the available information, because natural language text is often not well-formed and is interleaved with languages with different syntaxes: code fragments, stack traces, patches,* etc.

For these reasons, we must be aware of the structure of email content, filter irrelevant information, and use appropriate techniques for exposing and exploiting the significant data.

In this part of the thesis we present our work toward this goal: We present mining techniques aimed at giving a structure to the unstructured, noisy, and mixed-language content of development emails. These techniques are built incrementally, from the simplest but less effective, to the most sophisticated, flexible, and effective ones.

*In Chapter 7, we present our first step, in which we devised text-matching based techniques able to separate source code from natural language. Although results are promising, these techniques cannot be used to understand the meaning of the recognized lines of code. For this reason, in Chapter 8, we shift our attention to island parsing [136], and use this concept to devise a parser for structured content embedded in natural language. We present some applications of this parser to support software understanding and development. In the same chapter, we also present our implementation for a flexible and extensible island parsing framework. Finally, in Chapter 9, we present our approach to merge island parsing and machine learning to correctly recognize the different kind of "languages" (*e.g.*, stack traces, source code, and natural language) used in emails.*

Chapter 7

Detecting Lines of Source Code in Development Emails

When conducting the experiment described in Chapter 6, we realized that emails are often composed of different languages: Development emails pertaining to a software system report parts of text written in other languages, especially source code snippets or stack traces. These languages are more structured than natural language and cannot be processed with the same tools. In this chapter, we focus on simple textual techniques to detect source code fragments in natural language text.

7.1 Overview

Reliably extracting valuable information from the content of emails is a non-trivial task: Not only are emails written in free-form natural language, but they can also contain noise and interleaved structured fragments that makes it difficult to retrieve the relevant data. As a first step toward the exploitation of email content, in this chapter, we devise lightweight techniques to detect source code fragments in the content of emails, on the basis of simple text inspections, exploiting characteristics of source code text.

We evaluate the accuracy of our techniques through a benchmark we manually built. We took a statistically significant sample of emails pertaining to five unrelated Java OSS systems.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We identify the importance of identifying the structured content in email data.* The introduction of this part and this chapter explains why properly parsing email data is essential for extracting valuable information.
- *We devise and evaluate lightweight techniques that detect source code fragments in emails.* Our techniques exploit simple lexical characteristics of source code text embedded in development emails.
- *We produce a benchmark for evaluating the identification of source code lines in development emails.* Our benchmark features sets of sample emails, randomly extracted from five unrelated Java OSS systems, which we manually read to label structured fragments. It includes more than 1,800 manually labelled emails.

Structure of the chapter. In Section 7.2 we explain how we set up our benchmark and present the infrastructure we devised to support its creation. In Section 7.3 we detail the different approaches we tested. In Section 7.4 we discuss how they perform with respect to our benchmark. In Section 7.5 we discuss the related work. Section 7.6 concludes summarizing our findings.

7.2 Benchmark and Evaluation

Since no previous benchmark has been devised for the problem we tackle with our work, besides presenting and discussing a set of techniques, we also create a statistically significant, and publicly available benchmark, against which to verify them. Our benchmark for extracting source code fragments can be employed for further analysis of different techniques. The obtained results can be compared to show the strengths and drawbacks of several approaches on the same data set. We designed this benchmark to require no special infrastructure to be used and to be easily extensible with additional data.

7.2.1 Subjects of the experiment

Table 7.1: The software systems considered for the benchmark

System	Emails			
	Population		Sample	
	Size	Filtered	Size	With Code
ArgoUML	24,876	24,876	379	48
Freenet	22,095	22,095	378	35
JMeter	21,637	9,810	370	105
Mina	18,565	21,869	374	101
OpenJPA	14,992	6,328	363	97

Table 7.1 shows the five Java OSS systems that we considered to create our benchmark. We selected unrelated software systems emerging from the context of different free software communities, *i.e.*, Apache, ArgoUML, and Freenet. The development environment, the usage of the mailing lists, and the development paradigms are all likely to differ among the systems, providing a good test for the adaptability of our lightweight approaches to a wide variety of systems, and helping to assess their effectiveness.

Even though all the systems offer multiple mailing lists that can be analyzed, we focus on the development mailing lists, as they provide the highest density of information related to software development. We excluded from our benchmark messages automatically generated by the bug tracking system and the revision control system, since they contain only a reduced amount of natural language text. In the exceptional case of JMeter, we decided to also include part of the messages generated by the revision control system, to see their effect in our experiments.

The section `Population` of Table 7.1 provides details on the population size, and the number of emails in the set after filtering out the automatically generated ones.

Emails sample set size: Since we could not afford to manually annotate the entire set of nearly 75,000 emails, we extracted a sample. Due to the lack of any knowledge about the considered mailing lists, we employ simple random sampling [193] to extract the emails to be included in our benchmark. To determine the number of emails that must be sampled from the populations, we used Equation 4.1, already detailed in Section 4.2.1. We took the standard confidence level of 95%, and error (E) of 5%. This resulted in the values for the sample sets reported in Table 7.1. If source code fragments are present in the $f\%$ of the sample set emails, we are 95% confident they will be present in the $f\% \pm 5\%$ of the population messages. This only validates the quality of this sample set as an exemplification of the populations, and is not related to the *precision* and *recall* values presented later.

7.2.2 Benchmark creation

To evaluate our source code extraction methods, we manually built the benchmark by reading all the emails and annotating them with the source code fragments they contain. We inspected the entire sample set, and then randomly selected and re-inspected 10% of the emails to verify the quality of the annotations. Figure 7.1 shows the excerpt of an email with source code.

The blue parts are those we would have marked as source fragments in our benchmark. We consider the method call in line 2 as valid (it is actual source code that can be part of a method), but we do not consider the class, package, and method names, written in italic, in lines from 3 to 5, as they are part of the discussion and are simply used as names. We exclude commented lines of code (Line 13).

We extended the Miler Game (see Section 3.3), to assist this task. We created a `code snippets` panel (Point 1) to contain the selected fragments from the email. The user can add detected source code fragments by copying and paste them in the appropriate text area in the `code snippets` panel. We also enhanced the `Main` view (Point 2) so that the user can select the chosen fragments with the mouse and using a keyboard shortcut to add them in the annotations.

Despite the repetitiveness of the annotating task, we decided not to add features that could have influenced the results. For example, it would have been possible to highlight pieces of text containing Java keywords. However, this could have influenced the email reader, who could have only skimmed the email content in search of highlighted text, without checking the meaning.

7.2.3 Evaluation

To evaluate the techniques to detect documents and lines containing source code fragments, we use the IR metrics presented in Section 4.2.4: *precision* (Equation 4.2), *recall* (Equation 4.3), and *F-measure* (Equation 4.4).

To assess the effectiveness of our approach in extracting fragments, we computed the *Levenshtein distance* [123] line by line, between the text labeled as source code in the benchmark and the extracted fragments. This function, also called *edit distance function*, outputs the minimum number of changes needed to transform one string into another. The allowed transformation operations are deletion, insertion and substitution, and they are given the same unitary cost. As an example we consider the first source code fragment labeled in line 2 of Figure 7.1 (replicated in Figure 7.3 for readability): `add(LabelledLayout.getSeperator());`.

```
Hi Bob,
 I have used swidget version add(LabelledLayout.getSeperator()); from
org.argouml.uml.ui.LabelledLayout earlier and it worked fine.
There is another class LabelledLayout in org.tigris.swidgets that has
    method
getSeperator(), but it also does not work.
However, after transfer to new ArgoUML version there was no error
in code, but elements were not arranged in two columns any more.
Here is part of the code I have implemented:

import javax.swing.ImageIcon

private static String orientation =
 Configuration.getString(Configuration
   .makeKey("layout", "tabdocumentation"));

//make new column with LabelledLayout
add(LabelledLayout.getSeperator());

consequences = new UMLTextArea2(
   new
UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));

Could you help me, please?

Thanks,
Zoran
```

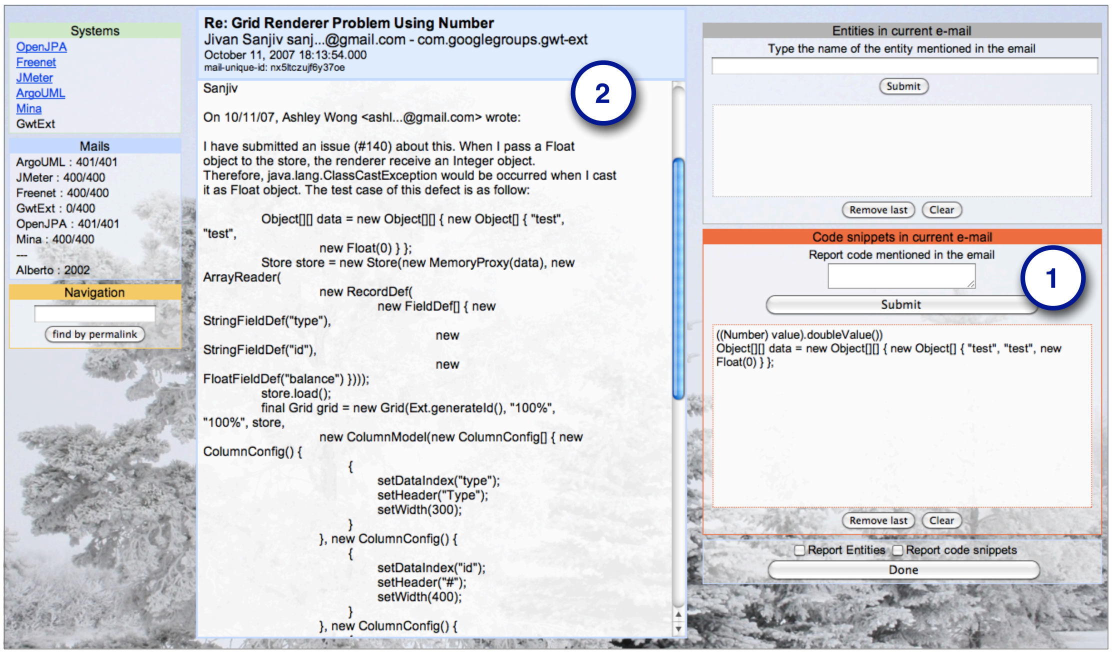
Figure 7.1: An email excerpt containing source code fragments

If we evaluate Levenshtein distance between this fragment and another candidate we obtain 0 if the two strings are identical, otherwise a positive number that increases linearly with the number of operations required to transform the candidate in the correct string. For example, the distance between this string and "*version add(LabelledLayout.getSeperator*" is 12.

Other edit distance functions, such as the *Damerau-Levenshtein distance* [57] and the *Hamming distance* [91], are not appropriate for our task as they add unnecessary operations (such as transposition of two adjacent characters) or do not provide enough flexibility, *e.g.*, the Hamming distance only applies to strings of the same size.

7.3 Experiments

We first tackle the problem of classifying emails that contain source code, then we move to the line level, and we conclude by showing how our methods can extract source code fragments. We begin from the techniques based on the simplest intuitions and we proceed to others based on more refined concepts.



ArgoUML : 401/401
JMeter : 400/400
Freenet : 400/400
GwtExt : 0/400
OpenJPA : 401/401
Mina : 400/400

Alberto : 2002

Re: Grid Renderer Problem Using Number
Jivan Sanjiv sanj...@gmail.com - com.googlegroups.gwt-ext
October 11, 2007 18:13:54.000
mail-unique-id: nx5ltczujf6y37oe

```
Sanjiv

On 10/11/07, Ashley Wong <ashl...@gmail.com> wrote:

I have submitted an issue (#140) about this. When I pass a Float
object to the store, the renderer receive an Integer object.
Therefore, java.lang.ClassCastException would be occurred when I cast
it as Float object. The test case of this defect is as follow:

        Object[][] data = new Object[][] { new Object[] { "test",
"test",
                new Float(0) } };
        Store store = new Store(new MemoryProxy(data), new
ArrayReader(
                new RecordDef(
                        new FieldDef[] { new
StringFieldDef("type"),
                                new
StringFieldDef("id"),
                                new
FloatFieldDef("balance") })));
        store.load();
        final Grid grid = new Grid(Ext.generateId(), "100%",
"100%", store,
                new ColumnModel(new ColumnConfig[] { new
ColumnConfig() {
                    {
                        setDataIndex("type");
                        setHeader("Type");
                        setWidth(300);
                    }
                }, new ColumnConfig() {
                    {
                        setDataIndex("id");
                        setHeader("#");
                        setWidth(400);
                    }
                }, new ColumnConfig() {
```

```
((Number) value).doubleValue())
Object[][] data = new Object[][] { new Object[] { "test", "test", new
Float(0) } };
```

Figure 7.2: The Miler Game extended to support email annotation of source code fragments

```
Hi Bob,
 I have used swidget version add(LabelledLayout.getSeperator()); from
org.argouml.uml.ui.LabelledLayout earlier and it worked fine.
```

Figure 7.3: An excerpt from Figure 7.1

7.3.1 Classification of emails including source code fragments

Special characters and keywords

Special characters (*e.g.*, semicolon, curly brackets) and reserved keywords (*e.g.*, *public*, *static*) are fundamental tokens with special meanings to the programming language, and are necessary to write the source code of any Java system. Even though some keywords are common dictionary words (*e.g.*, `for`) the presence of a high number of occurrences of keywords or special characters in a natural language text can be an evidence of an enclosed source code fragment. However, the length of the email content could influence the necessary number of occurrences to distinguish emails with source code fragments from those without them. Thus, we devised two different approaches to classify emails: 1. According to the number of occurrences of either Java keywords or Java special characters, and 2. according to keywords or special characters frequencies.

If the number of occurrences of keywords, or their frequencies, is above a certain threshold, we classify an email as containing source code fragments. We evaluate the results using several thresholds, to verify whether an optimal value can be defined.

Implementation. The occurrences of Java keywords in an email can be counted by dividing its content in words through any space separator (*e.g.*, end of lines, blanks) and summing one for each keyword occurrence. For the special characters, we do not divide the text in words, but we only count the occurrences of characters. To compute the frequencies we divide the number of occurrences by the total number of words or characters.

Results. Figure 7.4 shows the F-Measure results among all the systems, when considering occurrences of keywords. The maximum values obtained vary significantly between systems: from 0.31 for Freenet up to 0.63 for JMeter. The best threshold spans between distant values: 5 and 27. As an example, Figure 7.5 details OpenJPA showing precision, recall, and F-Measure.

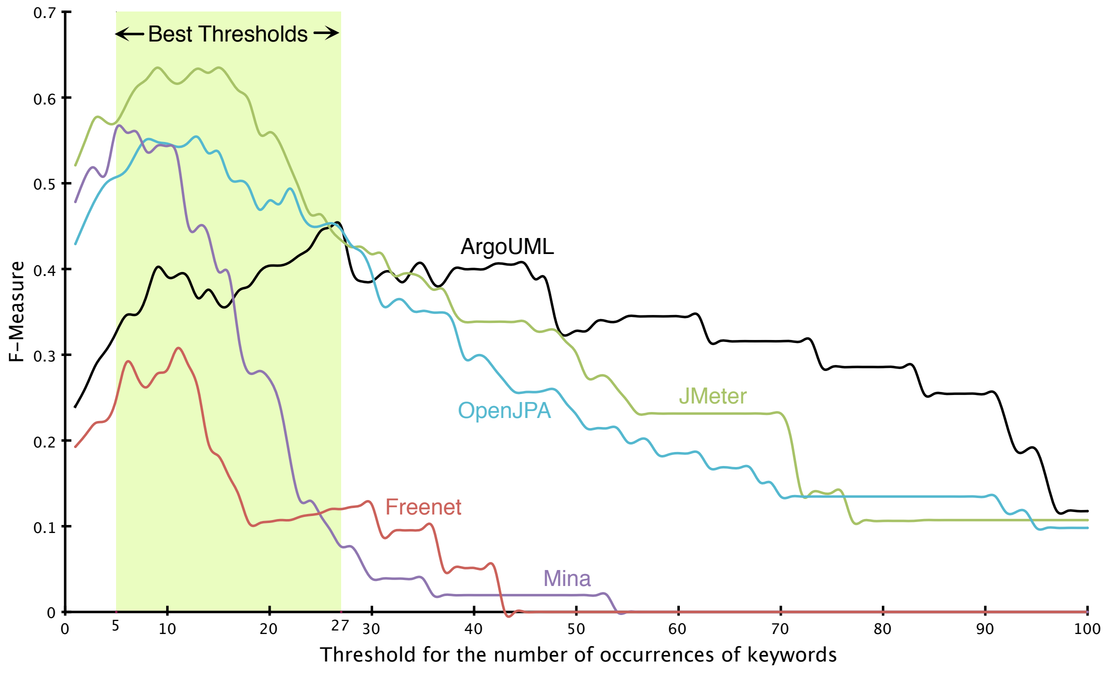


Figure 7.4: Email classification on *occurrences* of keywords

Precision and recall trade off one against the other, but a very low value in one of them does not automatically guarantee a extremely high value for the other: Both of them can be low. The relevant feature of this simple approach is that, by varying the threshold, we can obtain either almost perfect recall or perfect precision, according to our needs.

We obtained similar but more consistent results using special characters occurrences as our discriminator: the best F-Measure values vary between 0.50, for Freenet, and 0.63, for JMeter. The best threshold spans between less distant values: 200 and 600 occurrences of special characters. The curves have trends equivalent to those in Figure 7.4 and Figure 7.5.

Figure 7.6 shows the F-Measure values when considering frequencies of characters. In this case, the best thresholds span between more distant values and the F-Measure values do not show significant improvements compared to the simple occurrence count. For the single system, the curves still show a trend near to those in Figure 7.5.

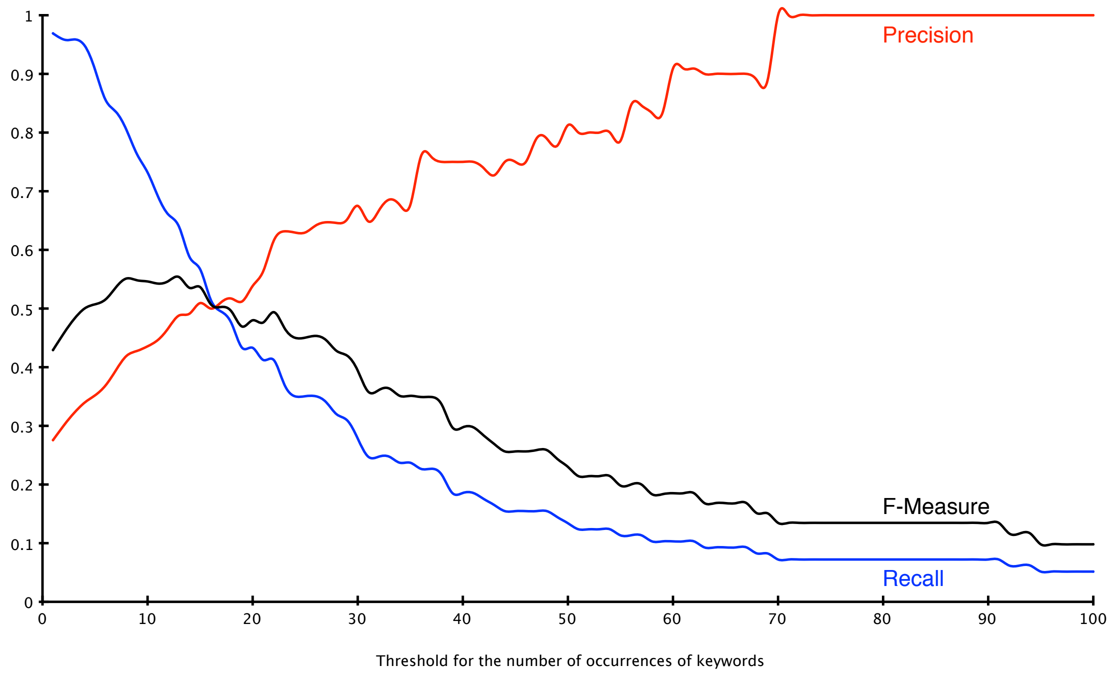


Figure 7.5: OpenJPA: email classification based on *occurrences* of keywords

End of lines

Taking the lines in Figure 7.7 as an example (corresponding to the source code lines in Figure 7.1), we note a peculiarity present in many programming languages (*e.g.*, JAVA, C, PERL): The developer must end each statement with a semicolon. In the email, we note that this happens in lines 9,12,14,17, and also in the code enclosed in line 2.

Based on this intuition, this approach verifies whether the text contains lines whose last character is a *semicolon*. Since a natural language text line does not often end with such a character, it can be a significant hint on the presence of source code fragments. To build a more comprehensive approach, we also consider that a source code line can end with a curly bracket, mainly used to open or close a block. This approach can be parameterized on a threshold that represents the number of lines that must end with the special convention.

Implementation. Even though this approach still classifies emails and not lines of code, we consider the presence of peculiar lines in the text. The implementation consists in analyzing the email content line by line and verifying if the last character is a semicolon or a curly bracket.

Results. Figure 7.8 shows the F-Measure for each system: the best values are significantly better than the previous approach: They vary from 0.74 to 0.92. Moreover, the best threshold is a single value (*i.e.*, one) that is the same for all the systems. Two lines ending with a semicolon are always the best indicator of the presence of source code fragments with this approach.

Taking FREENET as a sample system (Figure 7.9), we show that the approach can quickly achieve the maximum precision, while maintaining the recall value higher than 0.60. All the systems

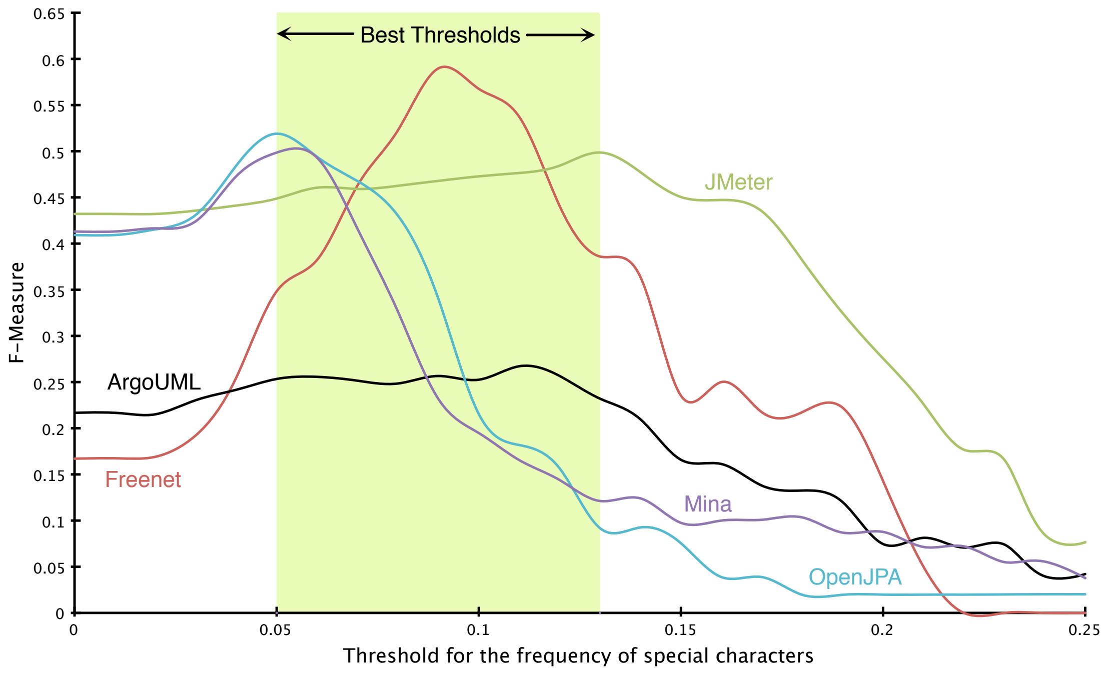


Figure 7.6: Email classification based on *frequencies* of special characters

show similar results. Even in the worst case, ArgoUML, at the second step of the threshold the precision is 1 and the recall is higher than 0.50.

End of lines and regular expression

Even though the previous approach reaches significantly high values, the recall can still be improved. Analyzing the false negatives, we noted that the method does not consider a common pattern in the Java programming language: the method call pattern. This pattern usually ends with a semicolon, but a method call can be split in multiple lines (as can be seen in line 11 of Figure 7.7), or used without semicolon in a stack trace. Our intuition is to raise the recall by checking this pattern in lines without a final semicolon.

Implementation. This approach extends the previous one, adding the check on the method call pattern. If a line does not end with any character specified in the previous method, it checks if it matches the method call pattern. The most effective technique to match such a regular pattern is using regular expressions. We implemented our regular expression, according the IEEE POSIX Basic Regular Expressions (BRE) standard, as in the following code:

```
(.*)
([[:alnum:]]+\.)+
([[:alnum:]]|<[[:alnum:]]+>)+
\(
(.*)
```

```
   [...]
 3 add(LabelledLayout.getSeperator()); from
   [...]
 9 import javax.swing.ImageIcon

10 private static String orientation =
11  Configuration.getString(Configuration
12    .makeKey("layout", "tabdocumentation"));
   [...]
14 add(LabelledLayout.getSeperator());

15 consequences = new UMLTextArea2(
16    new
17 UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));
   [...]
```

Figure 7.7: Lines with source code fragments in Figure 7.1

Lines 1 and 5 allow the match to be found without requirements on characters that follow or precede it. Lines 2 to 3 require one or more occurrences of a character followed by a single dot, *e.g.*, `foo.`, `boo.`, `boo.foo.` are valid sequences. After this pattern, there must be another alphanumeric string enclosed, or not, in angle brackets, *e.g.*, `foo`, or `<foo>`. Finally, there must be an open parenthesis (line 4). This regular expression correctly matches the example code in line 11, Figure 7.7. Such code would have been matched, even though the part after the open parenthesis was spread out over multiple lines.

Results. Table 7.2 shows the results for each system. We report the results obtained with the threshold of one, which is still the most effective, confirming the previous results. With this addition we included cases that were not correctly retrieved by using only the checking on the end of line. We significantly increased the recall, at the cost of a very small decrement in the precision. Overall, the results are promising: On average our lightweight approach retrieves 85% of the emails with source code, and correctly classifies 94% of the cases.

Table 7.2: End of line and regular expression approach results

System	Precision	Recall	F-Measure
ArgoUML	0.92	0.71	0.80
Freenet	0.97	0.91	0.94
JMeter	0.91	0.96	0.94
Mina	1.00	0.80	0.89
OpenJPA	0.89	0.88	0.88
Average	*0.94*	*0.85*	*0.89*

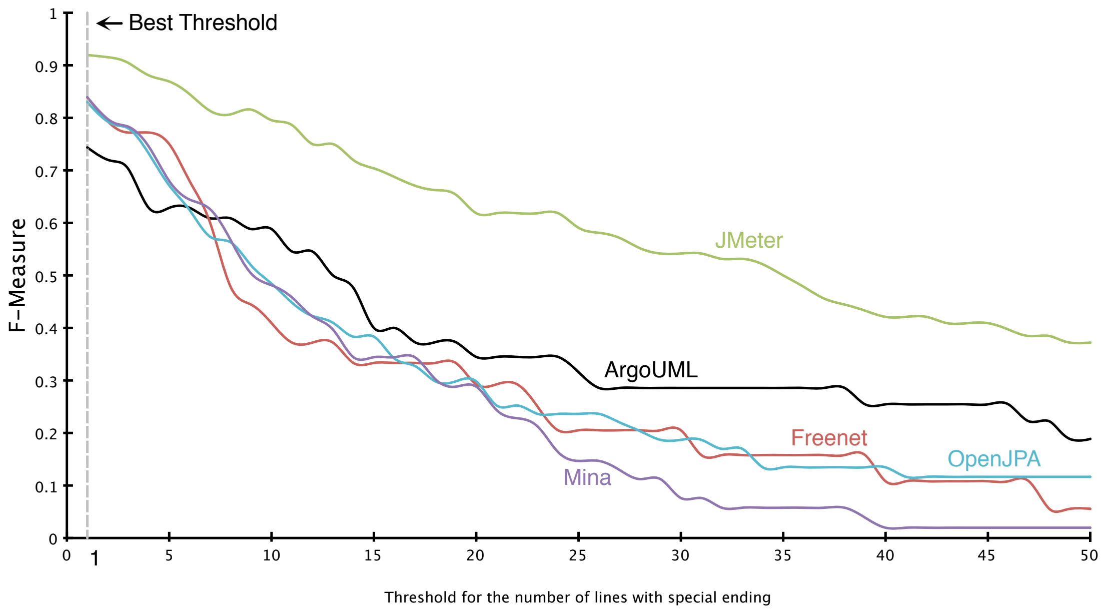


Figure 7.8: Email classification based on end of lines

7.3.2 Classification of text lines including source code fragments

Special characters and keywords

Our first method, which uses occurrence or frequency of special characters and keywords on the complete email content, has the advantage of retrieving *all* the emails with source code fragments, and still providing results with reasonable precision. This approach uses the same idea on the finer granularity of lines: a line having many occurrences, or a high frequency, of keywords or special characters probably contains a code fragment. We apply the classification on different thresholds to find the optimal one.

Implementation. This approach has a similar implementation to its equivalent for email classification, except that we split the text into lines and we count keyword and special character occurrences *per line*. We divide the occurrences by the number of words in the line to obtain frequencies. If a value passes the threshold, we classify the line positively.

Results. Table 7.3 shows the result obtained using the occurrence of characters for the line classification. It reports both the value with a threshold of one, which provides the best recall, and the threshold for the best F-Measure (if different from one). We note that the best threshold values vary less than in the approach for the email classification: between 4 or 5 for all the systems, except for JMeter. The results obtained using the frequency of special characters are similar but have lower values. Interestingly, for the line classification, both the occurrence and the frequency of keywords produce significantly lower results. Looking at Figure 7.7 we understand the reason: Eight lines out of the nine with source code fragments include, at least, one special character, while only four lines contain a keyword.

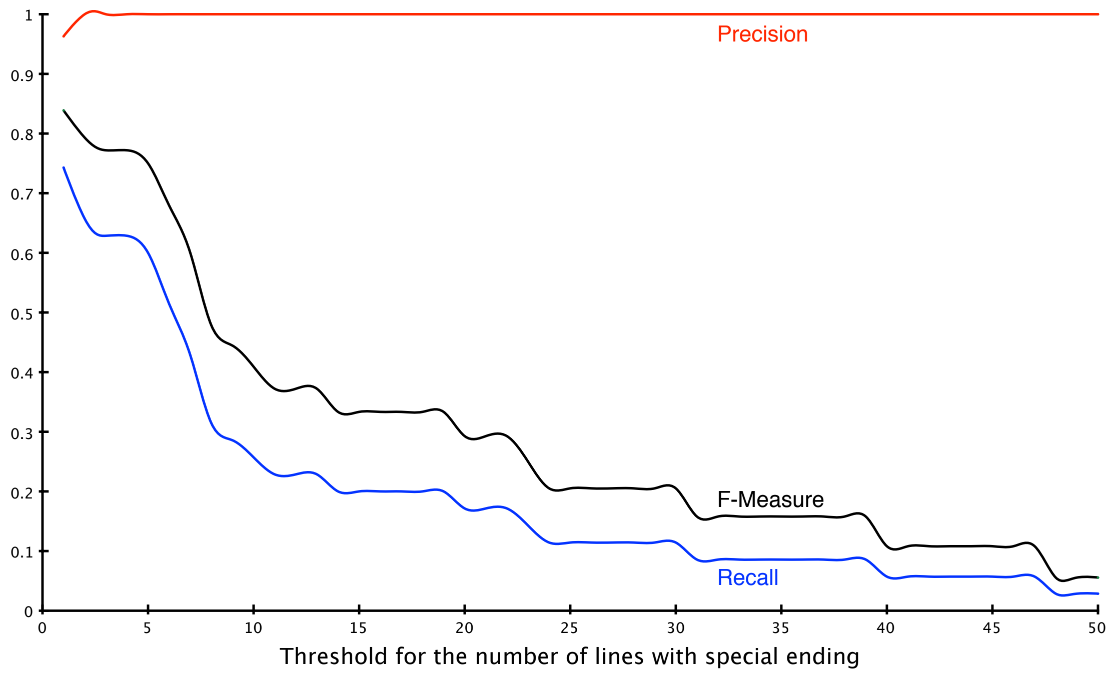


Figure 7.9: FREENET: email classification based on end of lines

End of line and regular expression

We use the approach for text classification for line classification: Each line that ends with a semicolon, a open or closed curly bracket, or that matches the regular expression described previously, is classified positively. The presence of code comments in a complete text can be evidence of a larger source code fragment, for this reason, we did not remove comments in the email classification approach. However, we do not consider comments as actual source code fragments, thus we remove them from lines during the classification. A threshold is not necessary: either a line does or does not respect the conditions. For the OPENJPA system, we also consider annotations as code fragments, since they are a relevant aspect of that Java persistence library.

Implementation. The implementation is similar to the complete text approach, but it classifies line by line, instead of the whole content. As a first pass, from each line, we remove the leading and trailing whitespace and the comments, *i.e.*, the text after `//`, or `/*`, and the text before `*/`. We detect OPENJPA annotations by checking whether the first character of a line is a `@`.

Results. Table 7.4 shows the results obtained, by system. The approach provides a very high precision (0.93 on average), keeping a substantial recall. For OPENJPA, the recall below the average is caused by the fragmentation of email lines: Many of the words are truncated and split over different lines. A text normalization [185] would alleviate this problem, probably providing the approach with results obtained for the other systems.

Table 7.3: Line classification by occurrences of characters

System	Threshold	Precision	Recall	F-Measure
ArgoUML	1	0.11	1.00	0.20
	4	0.19	0.62	0.29
Freenet	1	0.05	0.99	0.09
	5	0.17	0.40	0.24
JMeter	1	0.22	0.98	0.35
Mina	1	0.14	0.98	0.25
	4	0.30	0.58	0.40
OpenJPA	1	0.16	0.95	0.28
	4	0.25	0.66	0.36
Average	*1*	*0.14*	*0.98*	*0.24*
	best	*0.22*	*0.65*	*0.33*

Table 7.4: Line classification by end of line and regular expression

System	Precision	Recall	F-Measure
ArgoUML	0.93	0.89	0.91
Freenet	0.94	0.88	0.91
JMeter	0.97	0.86	0.91
Mina	0.90	0.84	0.87
OpenJPA	0.93	0.73	0.82
Average	*0.93*	*0.84*	*0.88*

Beginning of block

Even though the previous approach results meet a high target, we manually inspected all the false negatives to increase the recall. Lines with a class or method declaration are the most common problem: Methods and classes are usually defined in a single line, *e.g.*, `public class Foo()`, however, the curly bracket that starts the subsequent block is in a new line. Since our previous approach does not consider these cases, we also check them. The simple intuition is to check whether a line begins with a keyword: We see an example of this in Figure 7.7, line 10.

Implementation. The implementation, similar to the previous one, now also verifies whether the first word is a keyword.

Results. As expected, this approach increases the recall value of all the systems, of at least two points (Table 7.5). The high values already obtained with the previous approach, make it interesting for the practitioner who wants more matches, with a lower precision.

Table 7.5: Line classification by end of line and regular expression

System	Precision	Recall	F-Measure
ArgoUML	0.74	0.91	0.81
Freenet	0.60	0.90	0.72
JMeter	0.94	0.91	0.93
Mina	0.80	0.89	0.85
OpenJPA	0.77	0.74	0.75
Average	*0.77*	*0.87*	*0.81*

7.3.3 Source code extraction

Considering the high results that our lightweight approaches achieved on line classification, we conducted a statistical analysis on the benchmark to determine whether devising additional methods for a finer code extraction is necessary in practice.

Considering all the lines labeled as containing source code fragments in the benchmark (*i.e.*, 11,978), we evaluated the Levenshtein distance of the complete content from the source code part. More than 7,664 lines had a distance larger than three (*i.e.*, more than three operations are necessary to transform the content in the source code). Analyzing these lines, we noted that they are mainly composed of lines similar to those reported in Figure 7.10.

```
1      remove(LabelledLayout.getSeperator());
2   at org.apache.maven.Maven.doExecute(DefaultMaven.java:336)
3   + add(LabelledLayout.getSeperator());
4   - remove(LabelledLayout.getSeperator());
```

Figure 7.10: Common lines with source code distant from the content

In line 1, the source code does not include leading and trailing white spaces, which are present in the complete content. Line 2 starts with the `at` word used in stack traces, which is not source code. Lines 3 and 4 are part of a patch and start with `+` or `-` to mark added or deleted lines.

Removing these special cases and recomputing the Levenshtein distance, only 378 lines remained: the 3% on the total number of lines with code fragments. Since we considered statistically significant sample sets for the creation of our benchmark, this 3% ratio is a value that is to be found also in the whole mailing list populations.

Simply removing the special starting characters in Figure 7.10, and trailing and leading whitespace, we can use the same approaches explained for the line classification and obtain the perfect extraction of the source code for all the lines, except -at maximum- for those 3% that also include other not relevant characters. Since we want to maintain our methods fast and simple, we deem this result as acceptable.

7.4 Discussion

We presented a number of approaches to detect emails and lines that contain source code fragments. Table 7.6 summarizes the average effectiveness of the methods over the 5 systems, in terms of precision, recall and F-Measure.

Table 7.6: Average effectiveness of detection methods

Method	Precision	Recall	F-Measure
email classification			
Special characters and keywords	0.24	0.97	0.37
End of lines and reg. exp.	0.94	0.85	0.89
line classification			
Special characters and keywords	0.14	0.98	0.24
End of lines and reg. exp.	0.93	0.84	0.88
Beginning of block	0.77	0.87	0.81

Since the methods vary with respect to the precision and recall they achieve, choosing the most appropriate method depends on the relative weight of precision and recall, for a given task. If practitioners want to retrieve most of the source code, at the cost of many email lines without source code (*i.e.*, they want a very high recall at the price of a lower precision) the best method is the one based on the occurrences of special character and keywords. On the other hand, if one is interested in correctly retrieving only documents with source code (*e.g.*, for time reasons), at the price of not retrieving all of the documents (*i.e.*, favoring precision over recall), then "end of lines with regular expression" is the most appropriate method. It should be also selected when precision and recall have the same importance, since it offers the highest F-Measure. When good overall effectiveness (indicated by the F-Measure) are required, the mixed method based on end of lines, regular expression and keywords at the beginning of the line, is the best performer.

Comparison with similar approaches. In Section 7.5 we reported two approaches tackling a similar problem, one by Bettenburg *et al.* [30] and one by Tang *et al.* [185]. Bettenburg *et al.* detected bug reports containing source code fragments using an approach based on island parsing. Their technique reached results better than ours, with a precision of 0.98 and a recall of 0.99, an almost perfect classification. However, it is difficult to compare the results of the approaches, as they are applied on different data sets, with different characteristics (bug archive and development mailing list). Moreover, our methods have two main benefits over their approach: (1) they are faster and more scalable than island parsing and (2) they work correctly also at the line level.

Tang *et al.* applied a technique based on machine learning to detect source code fragments in a data set similar to ours but larger. In terms of precision, we reported similar results: they reached a precision of 0.93 and our best is 0.94 at the email level and 0.93 at the line level. We achieved better results in the recall value: 0.84 against 0.72.

Limitations. Due to different developer cultures, the style of emails in discussion lists may vary. To alleviate this, we considered five different open source software systems emerging from different communities, in which the usage, the participants, and the age of the mailing lists vary.

The main limitation of our experiment consists in considering only the JAVA programming language. However, we devised methods that are based on characteristics available in many different programming languages, *e.g.*, keywords, special characters, and peculiar end of lines.

We inspected accurately all the emails in the sample sets. However, since human beings are involved, there is the possibility that they made mistakes in the analysis. To avoid this problem we devised a web application to ease the task, and we manually re-inspected a relevant number of false negatives and false positives generated by our approaches during their construction, without finding errors.

7.5 Related work

Recovering Traceability Links

Our research is influenced by the seminal work of Murphy and Notkin [138]. They proposed a lightweight lexical approach, based on regular expressions, to extract models of a software system from different software artifacts. Software engineers can obtain consistent models from any kind of textual artifacts concerning software with such approach. The engineer must follow three steps: (1) define patterns (using regular expressions) that describe source code constructs of interest in a software artifact, *e.g.*, function calls or definitions; (2) establish the operations to be executed whenever a pattern is matched in an artifact being scanned; and (3) implement post-processing operations for combining information extracted from individual files into a global model. Although this approach is lightweight, flexible, and tolerant, the first step is non-trivial, especially when dealing with unstructured artifacts written in natural language, such as emails. Choosing the best approach to expose source code fragments of interest requires an accurate analysis of the advantages and drawbacks that the different regular expressions offer. Practitioners could find it difficult to perform this task through the iterative trial and error process proposed by Murphy and Notkin.

To our knowledge, little research has been performed to extract source code snippets from natural language written artifacts.

Bettenburg *et al.* proposed four filtering techniques to extract patches, stack traces, source code snippets, and enumerations from the textual descriptions that accompany bug reports [30]. The authors report results only on the effectiveness of their techniques in *differentiating* documents that contain source code snippets from those that do not contain source code. In this task, using island parsing [136], they reached almost perfect results, *i.e.*, a precision value of 0.98 and recall of 0.99. Since using a parser requires a high computational effort and scaling up to archives containing tens of thousands of documents can be problematic, we propose fast and lightweight techniques, based on regular expressions and pattern matching. In addition, we consider development mailing list archives as our natural language documents, which are more prone to noise not related to system development, if compared to bug reports. Finally, we evaluate the effectiveness of our techniques in detecting not only emails that contain source code, but also lines, and in extracting the fragments.

Bird *et al.* proposed an approach to measure the acceptance rate of patches submitted via email in open source software projects [35]. They have been able to classify emails containing source code patches. However, since email classification was not the focus of the work, the authors provided little information about their extraction techniques and no details on the benchmark they used to assess their effectiveness.

Dekhtyar *et al.* discussed the opportunities and challenges for text mining applied to software artifacts written in natural language [62].

Information Retrieval

Related work can also be found in the *information retrieval* and *data mining* field: A number of approaches in the information retrieval field are related to our work. Tang *et al.* addressed the issue of cleaning the email data for subsequent text mining [185]. They propose a cascaded approach to clean emails in four passes: (1) non-text filtering, (2) paragraph, (3) sentence, and (4) word normalization. In the first pass, what they consider non-text are actually email headers, signatures, and source code snippets. They randomly chose a total of 5,459 emails from 14 unrelated sources (*e.g.*, development newsgroups at Google) and created 14 data sets in which they manually labeled headers, signatures, quotations, and program codes. They used an approach based on Support Vector Machines (SVM) to detect source code fragments. They evaluated the effectiveness of the method at line level, and achieved reasonable results, *i.e.*, 0.93 in precision, and 0.72 in recall. These findings are promising, and in the data mining field, much research is devoted to extract information that has specific patterns using methods based on probabilistic and machine learning models (*e.g.*, Maximum Entropy Models [26] or Hidden Markov Models [24]). However, such approaches require more effort in the data collection, that could discourage an utilization by practitioners.

For this reason, we devised lightweight approaches based on simple methods to extract source code fragments from emails. Our approaches use lightweight and easy to implement techniques based on regular expressions that exploit intrinsic characteristics of source code elements.

7.6 Summary

In this work we tackled the issue of detecting and extracting source code fragments in development emails, at document level and at line level. We devised lightweight techniques that, on the basis of simple text inspections, exploiting characteristics of source code text, can detect source code fragments in emails, fast and with a high accuracy. A practitioner can precisely classify thousands of emails, even at run-time. We also proposed novel methods for classifying lines that enclose source code. Using refined approaches, based on those used for the complete document classification, our methods achieve performance higher than the ones previously obtained through complex machine learning techniques. Moreover, almost all methods we developed can be configured with a threshold parameter that allows choosing the best trade-off between precision and recall, according to the user's needs.

To assess our techniques, we created a statistically significant, easily extensible, and publicly available (`http://miler.inf.usi.ch/code`) benchmark: It features sets of sample emails, randomly extracted from five unrelated OSS systems written in Java, which we analyzed to label source code fragments. Using our benchmark, we conducted a statistical analysis of the email content and assessed that the vast majority of source code fragments are mentioned as lines separated from the natural language text. Our work shows that lightweight methods are to be preferred to heavyweight ones in identifying lines of source code from development emails.

Reflection. Even though our lightweight approaches to detect source code achieves good results in terms of effectiveness and practical performance, they specifically focus on recognizing code and can only be partially used in the context of a more comprehensive email text classification. For example, they merge lines of stack traces, patches, and actual source code, under the umbrella of code fragments. Although such an approach can be useful for certain system analyses, it generates a classification that does *not* allow one to (1) parse and extract meaning from the structured fragments detected, (2) distinguish lines written in NL; (3) recognize patch context and headers; (4) distinguish complete blocks of stack traces; (5) remove the non-relevant information. In the rest of this part, we present the methods we devised to address these concerns.

Chapter 8

Recovering Structured Fragments from Unstructured Data

In the previous chapter, we devised and evaluated lightweight techniques able to *identify* lines of JAVA code in emails. Since we found that the last character is a good indicator of the nature of a line, we implemented simple *lexical* rules, mainly based on pattern matching and regular expression. For example, we were able to detect most JAVA fragments by selecting lines ending with curly brackets or semicolons. By applying our previous approach on the text in Figure 8.1, we would identify that lines 12 and 14 are code.

```
 1  Because of problems with "argoHome" location, I imported
 2  java.net.URLDecoder and added the line
 3  URLDecoder.decode(argoHome); just below this one:
 4  public void loadModulesFromDir(String dir) in ModuleLoader.java.
 4
 5  Another problem in ModuleLoader:
 6  since the cookbook explains how to make a PluggableDiagram (that's
 7  exactly what I am doing, so I extend this class), I have not found
 8  where the JMenuItem returned by method getDiagramMenuItem() in
 9  PluggableDiagram is attached in Argo menus. It seems this is not yet
10  implemented, even though PluggableDiagrams implements Diagram.
10
11  So I have added those lines:
12  void append(PluggableDiagram aModule) {
13    ProjectBrowser.TheInstance
14       .appendPluggableDiagram((PluggableDiagram)aModule); }
14
15  Of course, such modifications must be reflected in ProjectBrowser.
```

Figure 8.1: Example document enclosing structured information

Although this result is precise, it is a simple binary *classification*: Code or non-code. After lines are classified as code, we would need an additional parser to first understand their content and meaning (*e.g.*, the whole block is the declaration of the method *append*) and then derive a model with the contained information. This is not feasible in practice, because our previous technique often loses the context (*e.g.*, line 13 is not recognized, thus we lose the method declaration context), and is not able to recognize code fragments embedded in natural language lines (*e.g.*, the methods in lines 4 and 8) and class names.

To derive correct and complete information from structured fragments in natural language artifacts, we need a more robust approach that allows the *identification*, the *extraction* and the *parsing* of the structured fragments occurring in artifacts written in natural language. In other words, we must be able not only to detect source code fragments, but also we must precisely determine the nature of those fragments (*e.g.*, decide whether a fragment is a method invocation or a class declaration), extract the contained information (*e.g.*, the name of the declared class), and derive models from them. In this chapter, we present and validate two approaches to accomplish this, which are the results of subsequent improvements in our research. To provide more evidence toward our thesis, we also show how these can be applied to support software understanding and development.

8.1 Overview

In software engineering, textual software artifacts (*e.g.*, emails) comprise natural language text that is interleaved with languages following a formal and structured syntax, like source code fragments, stack traces, patches, *etc.* Applying IR methods (*e.g.*, hierarchical clustering from Data Mining (DM), or Natural Language Processing (NLP)) without first recognizing structured fragments reduce the quality, reliability, and comprehensibility of the available information, because they are not expected to work with languages other than natural language; they have been proven limited or too laboriously tailored to the intricacies of the underlying data and intended use cases.

For this reason, an adequate analysis of artifacts containing both structured and unstructured fragments requires using different techniques for these two different classes. A better analysis exploits NLP or IR on relevant and well-formed natural language sentences and uses parsers on structured content, such as code or log snippets. To enable this, it is necessary to exactly filter out natural language sentences from structured data. Although structured elements are defined through formal grammars, this separation is non-trivial because structured elements are rarely complete and well-formed in natural language documents; instead, they may appear as incomplete, such as method definitions lacking their body.

This chapter contributes to the extraction of structured fragments in natural language software artifacts. We developed two approaches, named iLander and PetitIsland, based on the concept of island parsing [136], that can be used to *extract*, *parse*, and *model* structured data found within artifacts containing arbitrary text, such as development emails (but also forum posts, issue reports, requirements, *etc.*). Island parsing is the conceptual foundation of our approach: it describes parsing of interesting structures, the *islands*, from a "sea" of filtered strings. While using island parsing to separate natural language text from structured fragments is not new, we introduce two novel and effective approaches to implement island parsing in this context.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We identify the importance of* understanding *the structured content in email data*. We realize this cannot be achieved using a lexical approach and regular expressions, but it requires a full-fledged parsing approach, which we devise.

- *We devise* ɪLᴀɴᴅᴇʀ*, an island parser to recognize, extract, and model source code fragments immersed in natural language text.* ɪLᴀɴᴅᴇʀ is based on the ASF-SDF Meta-Environment [196], and we evaluate it extracting information from email data.
- *We produce a benchmark for evaluating the recognition, extraction, and modeling of* Jᴀᴠᴀ *content in email data.* We create our benchmark by reading and labeling sample emails from four unrelated Jᴀᴠᴀ OSS systems. It comprises 188 labelled emails with structured fragments.
- *We use* ɪLᴀɴᴅᴇʀ *to conduct a number of software evolution analyses.* We reconstruct the model of a software system from its emails and we detect salient moments in its history.
- *We created* PᴇᴛɪᴛIsʟᴀɴᴅ*, a* flexible *and* extensible *framework for building and composing island parsers.* PᴇᴛɪᴛIsʟᴀɴᴅ is written in Sᴍᴀʟʟᴛᴀʟᴋ and is based on the parser generator PᴇᴛɪᴛPᴀʀsᴇʀ. We evaluate it by extracting source code fragments from Stack Overflow posts.
- *We adapt and improve a previously published benchmark to assess the recognition of source code fragments in Stack Overflow posts.* The renovated benchmark comprises 188 posts embedding more than 350 code fragments.

Structure of the chapter. Section 8.2 illustrates the grammar concepts used in the chapter and the parsing techniques that we used to implement our approach. Sections 8.3 to 8.6 present our first approach, ɪLᴀɴᴅᴇʀ: Section 8.3 details its rationale and implementation, Section 8.4 its validation, Section 8.5 the modeling of source code data, and Section 8.6 its applications. Sections 8.7 to 8.9 present our second approach: PᴇᴛɪᴛIsʟᴀɴᴅ: Section 8.7 motivates its advantages over the previous approach and present its implementation, Section 8.8 describes its validation, and Section 8.9 discuss its applications. Section 8.10 illustrates the related work. Section 8.11 summarizes the contributions of this chapter.

8.2 Grammars and Island Parsing

In this section we introduce basic grammar and parsing concepts used in this chapter.

A *generative grammar*, or simply a *grammar*, is a formalism used to mathematically define a *language*. A grammar is composed of a set of *terminal symbols* (or simply *terminals*), a set of *nonterminal symbols* (or *nonterminals*), and a set of *productions*. Terminals and nonterminals constitute two disjoint sets. The set of terminals identifies the symbols that compose each possible string of the language to be defined. In programming languages, terminal symbols are all characters valid in the language; they typically include special characters, punctuation, brackets, *etc.*

By means of a grammar, language designers define a syntax through the application of *productions*, which essentially denote string rewriting. Each production is composed of a left-hand side (LHS) and a right-hand side (RHS): The former identifies a set of symbols that can be rewritten to the latter.

Programming languages are commonly defined using a particular class of grammars, *context-free grammars*, where the LHS contains only one nonterminal symbol. Each application of a production rewrites a nonterminal into a sequence of other symbols, which can be either terminal or nonterminal. The successive application of productions (*i.e., rewriting*) starts from a special initial nonterminal symbol and rewrites it until obtaining a string composed of terminals only. The resulting string is a valid string of the language. A *derivation* is a sequence of production applications that leads to a string of terminals. For programming language grammars, nonterminals are used to define the possible syntax of particular constructs, such as function declarations, invocations, statements, *etc.*

A *parse tree* can represent the syntactic structure of a string belonging to a language defined with a grammar. A parse tree is a rooted tree where all internal nodes are labeled with nonterminals, while leaves are labeled with terminals. A *parser* is an algorithm that produces a parse tree from a given string. Parsers are commonly produced by *parser generators*, which take a grammar as input and produce a parser for the language described by the grammar. Different parser generators support different classes of grammars with potential restrictions on the grammar structure.

If a grammar is *ambiguous*, multiple parse trees can be obtained from parsing the same given input string. An ambiguous grammar can produce a given string in different ways, through different sequences of productions, leading to different parse trees, i.e., to a *parse forest*. Disambiguation mechanisms can be applied to prune a parse forest and select a preferred parse tree, by using *priority mechanisms* between alternative productions. A typical example of ambiguous grammar is the following, for arithmetic expressions including sum, product and constants.

$$A \to \quad A+A \mid A*A \mid x$$

The grammar above is ambiguous for two reasons: (1) each rule describing binary operators can be parsed with different (left or right) associativity; (2) the grammar admits different parse trees where sums, for example, may have a higher priority than products. While different associativity for the same operator does not change the semantics of the evaluation of expressions, giving different priority to product and sum may produce wrong evaluations.

Figure 8.2 shows two parse trees for the expression `2*1+3` parsed with the grammar above; the parse tree on the left is wrong, since sum should not have a higher priority than product.

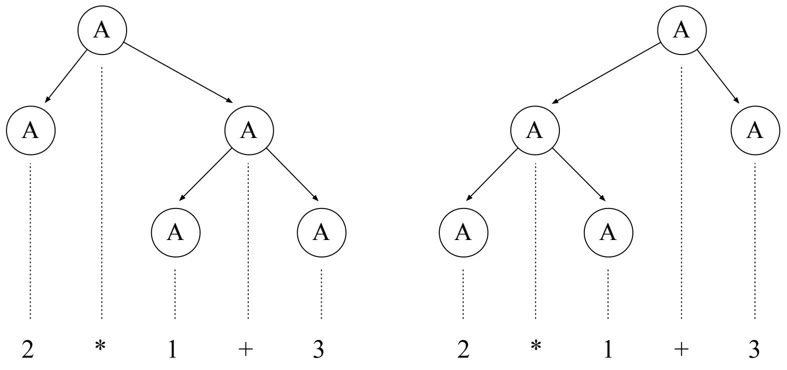


Figure 8.2: Two different parse trees for the expression `2 * 1 + 3`.

Most context-free grammar parser generators can handle ambiguities by providing ad-hoc techniques to select the right parse tree among a parse forest. In Yacc/Bison [158], for example, operator tokens like $+$ and $*$ appearing in ambiguous rules can be given priorities and associativity rules.

8.2.1 Island Grammars and Their Parsing

Our approach to extract structured content like source code from NL documents relies on *island grammars*. Island grammars are defined as "detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water)" [136]. In our case, structured fragments (*e.g.*, source code fragments) constitute the islands to be extracted, while the rest (*e.g.*, NL sentences) is considered as water. The grammar required to correctly parse arbitrary structured fragments, such as source code, may reach the full complexity of context-free languages in terms of expressiveness.

Other approaches for the extraction of source code from NL artifacts (*e.g.*, [30; 167]) used regular expressions to deal with the extraction, mainly for performance reasons. However, since programming languages are not regular languages, but typically contain recursive structures (*e.g.*, nested blocks) of context-free languages, these approaches are prone to providing low precision and recall. With regular expressions alone, it is theoretically impossible to correctly parse parenthesized expressions or nested blocks, which are present in almost every programming language. For this reason, a precise island parser for code fragments immersed in a sea of NL requires—at least—the capability of parsing context-free languages.

Parser generators for programming languages separate the *tokenization* phase from the parsing phase. At first, the input code is *scanned* with regular expressions to classify elements as either keywords, identifiers, operators, or other tokens like brackets. However, island grammars are inherently ambiguous, also at the token level. For example, it is impossible to determine, without its context, if the word `public` belongs to a NL sentence (*e.g.*, "*This is a* public *API.*") or to a structured element like a Java class definition (*e.g.*, `public class Tree();`). For this reason, the separation between tokenization and parsing in island grammars is not convenient, and may complicate the grammar structure for parsing. For this reason, island grammars prefer *scannerless parsing* techniques [136].

Our application of island parsing requires parsing techniques that handle ambiguous grammars. Few parsing algorithms support the full flexibility of context-free grammars with ambiguities, such as *Generalized LR* [192], with its scannerless version called *SGLR*, or the *Earley* [65] algorithm. In the next sections, we present our solution to implement island parsing using GLR parsing and a scannerless approach based on the SDF grammar definition formalism [197]. Sdf (Syntax Definition Formalism) is able to specify ambiguous grammars and provides specific techniques to parse such grammars and solve ambiguities.

8.3 iLander: The Parsing Approach

We base our implementation on the ASF-SDF Meta-Environment [196]. Among other things discussed later, we use it to define the context-free grammar necessary for our approach, by means of the syntax definition formalism (Sdf) [96]. We briefly outline the key features that we employ in our work.

By using Sdf, we define context-free grammars in a modular way, thus facilitating the derivation of an island grammar from any existing programming language grammar. Within Sdf, we can use a *Scannerless Generalized LR parser* (SGLR), which does not impose any restrictions on the grammar. This property is essential when parsing artifacts based on island grammars, which are ambiguous by nature. For *ambiguity management*, we make use of the disambiguation constructs, such as priorities, restrictions, or preference attributes to favor one particular production when several alternatives exist. Finally, we use *traversal functions* [195], which enable the analysis and rewriting of complex parse trees by focusing only on particular nodes of interest (*e.g.*, code fragments).

Input and Output. Our extraction process takes text files as input (*e.g.*, Figure 8.1) and outputs the set of structured fragments they contain (highlighted parts in Figure 8.1). For each fragment, our approach keeps track of the exact location (*i.e.*, the area) within the container file. An extracted structured fragment consists of a list with three elements (*e.g.*, Listing 8.1): (1) the corresponding nonterminal to which the fragment has been reduced, *i.e.*, the fragment type (lines 1, 4, and 7 in Listing 8.1), (2) the file name and the fragment's coordinates within it (lines 2, 5, and 8), and (3) the fragment content (lines 3, 6, and 9–11).

```
 1  ( CLASS_NAME :
 2    area-in-file("exampleDocument", area(2, 0, 2, 19, 57, 19)) :
 3    java.net.URLDecoder )
 3
 4  ( METHOD_INVOCATION :
 5    area-in-file("exampleDocument", area(3, 0, 3, 29, 96, 29)) :
 6    URLDecoder.decode(argoHome); )
 6
 7  ( METHOD_DECLARATION :
 8     area-in-file("exampleDocument", area(12, 0, 14, 62, 608, 131)) :
 9     void append(PluggableDiagram aModule) {
10       ProjectBrowser.TheInstance
11         .appendPluggableDiagram((PluggableDiagram)aModule); } )
```

Listing 8.1: Examples of extracted structured information

Grammar Notation. The grammar fragments used in this part are written in Sdf notation. It is important to underline that in Sdf, the left-hand side (LHS) and right-hand side (RHS) of grammar productions are swapped when compared to BNF-like notations. In other words, a production normally written in BNF as $lhs ::= rhs$, is denoted as $rhs \rightarrow lhs$ in Sdf. Consider for example the following Sdf production:

```
Modifier* MethodRes MethodDeclarator Throws? → MethodHeader
```

In this case, `MethodHeader` (at the right of →) is the LHS of the production and defines the syntax of a method header. Sdf also supports special symbols to define repetition and optional constructs. For example, the question mark after the `Throws` nonterminal in the aforementioned production notifies that the construct is optional, that is, it might not be present in a valid method header. Instead, the star in the `Modifier` nonterminal notifies that a valid method header can have multiple modifiers. Other productions define the syntax of the other constructs present in the production, like `Throws` constructs, *etc.*

8.3.1 Island Definition

Common programming language grammars describe different constructs at different abstraction levels, which range from simple identifiers and keywords to whole compilation units.

We can define an island as a piece of code that can be reduced to any possible nonterminal in the grammar. For example, an interesting fragment could be a piece of code that can be parsed and reduced to a nonterminal `MethodDeclaration` (*e.g.*, lines 12–14, Figure 8.1, repeated in Figure 8.3, are reduced to the block (lines 7–11) in Listing 8.1).

```
12 void append(PluggableDiagram aModule) {
13   ProjectBrowser.TheInstance
14      .appendPluggableDiagram((PluggableDiagram)aModule); }
```

Figure 8.3: Example lines from Figure 8.1 enclosing structured information

Ambiguities in Definitions. By considering every possible nonterminal as an island, we might introduce *ambiguities* that must be resolved and that could affect the performance and the effectiveness of the fragment extractor. For example, we consider the following syntax productions of the Java grammar:

```
Modifier* MethodRes MethodDeclarator Throws? → MethodHeader
Type   → MethodRes
```

The last production declares that any `Type` is also a valid `MethodRes`, which is a valid return type for a method in this context. If the aim is to extract both `Type` and `MethodRes` nonterminals emerging from water, a parser for the island grammar should be able to resolve the ambiguity between these two nonterminals: For example we should define the preferred among every possible equivalent alternatives. This ambiguity management degrades performance and should be avoided. We limited ambiguities from productions, and their resolutions, to a few cases, for example the extraction of some classes of valid identifiers, such as class and package names. In Section 8.3.2 we describe how we resolve ambiguities for this specific class of nonterminals by using naming conventions.

Productions. The choice of productions is related to the abstractions we want to derive from the fragmented information in an artifact. Since our aim is to derive structured information about a target software system, many possible fragments corresponding to some rules are irrelevant. For example, isolated expressions or generic statements seldom carry relevant information in terms of methods or classes of the system. Instead, they carry structural information if, and only if, they contain specific sub-operands (as in the case of expressions) or are specific statements. For example, in the valid Java expression `getInteger() / n`, the most interesting information is the method invocation. Thus, instead of choosing the nonterminal `Expression` as a valid source fragment, we choose only those subexpressions that carry information about the system structure, such as method and constructor invocations.

Incomplete Productions. Incomplete productions are a source of differentiation from a traditional programming language grammar. As an example, we consider line 4 in Figure 8.1 (repeated in Figure 8.4): It explicitly references a method signature but the body is missing.

```
public void loadModulesFromDir(String dir) in ModuleLoader.java.
```

Figure 8.4: Incomplete method declaration from Figure 8.1

The standard Java grammar includes only method definitions that are followed by either a semicolon (when they appear in interfaces) or the whole method body. By considering only the fragments that can be reduced to a nonterminal in the standard grammar, we might lose relevant information, such as the incomplete method declaration in Figure 8.4. For this reason, we also extract incomplete information corresponding to a subset of a production that does not reduce to any nonterminal in the standard programming language grammar.

Considering every possible incomplete production would result in another source of ambiguity, which in turn would affect the performance of the fragment extractor. Incomplete productions must be selected according to the kind of model that needs to be extracted from the artifact.

Since they add relevant structural information, our island grammars include incomplete productions of the nonterminals representing declarations of methods, constructors, and classes. Such incomplete productions do not require a final semicolon or a block with the body of the construct. For example, the following production has been introduced to extract incomplete constructor declarations:

```
Modifier* ConstructorDeclarator Throws? → IncompleteConstructorDeclaration
```

Our approach also extracts the entity declaration even in the case of a body which is incomplete or contains non valid fragments. For example, for a class declaration with a partial body, our method extracts a single fact for the declaration, as an incomplete class declaration, and parses the partial body as if it was a sentence of the island grammar.

```
 1  On 4/12/07 Bob wrote:
 2    [...]
 3  > public class Bicycle {
 4  >
 5  >  void changeCadence(int newValue) {
 6  >   cadence = newValue;
 7  >  }
 8  >
 9  >  void changeGear(int newValue) {
10  >   gear = newValue * 2;
11  >  }
12  Bob I believe you should change this method to fix the bug.
13  > }
```

Figure 8.5: Examples of incomplete class declaration with partial body

If we consider the document in Figure 8.5, the declaration of the class `Bicycle` is not correct, because line 12 is not a valid Java sentence. In this case, we prefer to loose the binding between the declaration and the elements of the partial body: We extract a single fact for the incomplete declaration of class `Bicycle` and parse the partial body as it was a sentence of the island grammar, thus extracting the two complete method declarations (*i.e.*, `changeCadence(int)` and `changeGear(int)`). A more complicated alternative is the *islands with lake* [136] approach that enables the extraction of bodies with water.

Another kind of incomplete fragments containing interesting structural information are *class relationships*. These express inheritance and implementation relations between classes and interfaces. For example, in line 10 of Figure 8.1, we find two potential class names separated by the keyword `implements`. From this, we can derive that `Diagram` is an interface, and there is an implementation relation between the two entities.

Additional Structured Data. Programming languages like Java are composed not only of the language definition, but also of additional components, such as the virtual machine, for which new entities are defined with a precise syntax. Such entities range from stack traces to simple file name requirements for compilation units. These entities, usually language-specific, carry interesting information about the system. Within the documentation produced during development phases, those external entities are usually present between code and natural language sentences, and thus they can be extracted as relevant structured facts. We chose to enrich the island grammar to support the extraction of Java stack traces and file names that follow the language naming conventions [184]. For example, we added the following production to support stack trace lines:

```
"at" (Name ".")? Identifier "." Identifier
"(" LexJavaClassFileName LexJavaExtension ":" DecimalIntegerLiteral ")" →
    JavaStackTraceLine
```

In this case, the two identifiers are always used to define the class name and the method name where the exception has been (re)thrown, while between the parentheses we can find a file name of a Java class containing the class definition. Instead, we structure a Java class name reference as in the following:

```
(LexPackagePath LexPathSep)? LexJavaClassFileName LexJavaExtension →
    JavaFileName
```

The above production describes a filename where the extension can be either `.java` or `.class`, and where the path is optional and might be an existing package (*e.g.*, in Figure 8.5, the last part of line 4 is recognized and reduced to this production).

Table 8.1 summarizes the nonterminals of the Java programming languages that we chose to extract as source fragments.

8.3.2 Ambiguity Resolution

Island grammars are inherently ambiguous: Water is defined as *anything* that is not an interesting fragment (*i.e.*, an island). Language definition systems supporting ambiguous grammars, like Sdf, provide constructs to resolve ambiguities. To choose between alternative derivations, we use two disambiguation keywords offered by Sdf:

- **Avoid.** The parser removes alternative derivations that have avoid at the top node, but only if there are no other alternative derivations with avoid at the top node.

Table 8.1: Source fragments recognized in iLANDER

Nonterminal	Description	Nonterminal	Description
CompilationUnit	Class declaration with package imports	JavaStackTraceLine	Stack Trace lines
ClassDeclaration	(In)complete class declaration	IfThenStatement	Conditional Blocks
MethodDeclaration	(In)complete method declaration	IfThenElseStatement	
ConstructorDeclaration	(In)complete constructor declaration	TryStatement	Try/Catch blocks
FieldDeclaration	Class field declaration	WhileStatement	
MethodInvocation	Method invocation	ForStatement	Loops
ConstructorInvocation	Constructor invocation	DoStatement	
JavaClassName	Java Class names	ClassRelationshipFragment	Implements/Extends relations
JavaFileName	Java File Names	Block	Alone blocks

- **Prefer.** The parser removes all other derivations that do not have prefer at the top node.

By using the *avoid* keyword for any reduction to water, we favor any other production against it, *i.e.*, we prefer islands. We exploit the same mechanism to give preference to some island structures. For example, consider the complete method declaration in Figure 8.3 (lines 12–14). As shown in Table 8.1, at the same level of `MethodDeclaration`, we have `Block` and `IncompleteDeclaration`. Thus, the construct in the example is ambiguous: It can be parsed either as a `MethodDeclaration` or as a sequence of `IncompleteDeclaration` plus `Block`. To disambiguate, we favor the solution that reduces to a single nonterminal, this is coherent with the aim of keeping the binding between the parts.

Additionally to basic ambiguity resolution techniques, such as the aforementioned ones, we support more complicated cases. Consider the fragment *"by method getDiagramMenuItem()"* (Figure 8.1, line 8, repeated in Figure 8.6).

```
8  where the JMenuItem returned by method getDiagramMenuItem() in
```

Figure 8.6: Ambiguous method invocation from Figure 8.1

The island grammar described in the previous section would select *"method getDiagramMenuItem()"* as a valid `IncompleteMethodDeclaration` fragment: *"method"* is a valid identifier, and thus a lexically valid return type for the JAVA grammar. However, JAVA naming conventions [184] prescribe that class names must be capitalized, thus, *"method"* violates the naming conventions. We exploit this to exclude those reductions to incomplete method declarations where the supposed return type is likely to be invalid.

For every extracted fragment, we define an ASF function, called `isValidSource`, which takes a source fragment as input and returns true iff the fragment is valid. The base case for that function is true for any fragment. For incomplete method declarations, we define `isValidSource` to return true iff the return type is `void`, a primitive type, or an identifier that respects to naming conventions. With the ASF syntax, the rule is declared with three different cases:

```
isValidSource(#IncompleteMethodDeclaration) = true when
  void #MethodDeclaration := #IncompleteMethodDeclaration
```

```
isValidSource(#IncompleteMethodDeclaration) = true when
   #PrimType #MethodDeclaration := #IncompleteMethodDeclaration

isValidSource(#IncompleteMethodDeclaration) = true when
   #Identifier #MethodDeclaration := #IncompleteMethodDeclaration,
   isAClassName(#Identifier) == false
```

Similar definitions are done for potential constructor declarations, which must respect valid naming conventions.

Going back to the previous example, the "*method getDiagramMenuItem()*" is not a valid method declaration, but it can be restructured as a method invocation fragment "*getDiagramMenuItem()*". We define an ASF transformation rule to translate what was parsed as a method declaration to a method invocation:

```
extractCodeFragments(#Source,#ExtractedSourceFragments)
                        = #ExtractedSourceFragments
(MethodInvocation : getLocation(#MethodDeclaration) :
   #Identifier()) when
  #IncompleteMethodDeclaration := #Source,
  #MethodRes #MethodDeclaration := #IncompleteMethodDeclaration,
  #Identifier() := #MethodDeclaration,
  isValidSource(#IncompleteMethodDeclaration) == false
```

The rule takes a fragment parsed as an incomplete method declaration violating the naming conventions, extracts the identifier corresponding to the method name, and produces a source fragment of type `MethodInvocation`. The location of the transformed fragment is the same as `#MethodDeclaration`, the nonterminal corresponding to the method name and its formal parameter declaration, which here is empty. This transformation must be applied only to methods without parameters.

8.4 iLander: Validation

We validate our approach by performing a case study with real world software systems. We consider three OSS systems developed in Java: ArgoUML, Freenet, and Mina (see Section 3.4 for more details on these systems). To improve generalizability, we picked systems from different domains that are developed by distinct free software communities.

8.4.1 Text normalization of emails

Due to the noisy nature of email content, we devised the following pre-processing text normalization phase.

Header and quotation removal. We remove the email metadata and the occurrences of the character > at the beginning of lines (used to mark different quotation levels).

Patch preprocessing. When communicating about implementation of entities, developers often exchange information in the form of code patches to show how parts of code have been changed. Figure 8.7 shows an example.

```
1  @@ -64,4 +72,7 @@
2        */
3       public void actionPerformed(ActionEvent event) {
4          super.actionPerformed(event);
5  -        new Import(ArgoFrame.getInstance());
6  +        if (ImporterManager.getInstance().hasImporters()) {
7  +             new Import(ArgoFrame.getInstance());
8  +        } else { LOG.info("Import sources dialog not shown"); }
9       }
```

Figure 8.7: Example patch

The common format for patches is the unified *diff* format[198], which uses a header that is not ambiguous with natural text (*e.g.*, line 1). New added lines are marked with a + sign (*e.g.*, lines 6–8), while removed lines have a - sign (*e.g.*, line 5). Lines without special signs give the context in the code. As the special signs can interfere with the parsing process, we perform a normalization process that 1. detects a patch fragment (using the header), 2. removes the lines marked as deleted (to avoid recovering what is explicitly no longer valid), and 3. removes the + signs at the beginning of the lines.

Stack trace normalization. When the execution of a Java application is interrupted by an unhandled exception, the program outputs a trace of the calling stack. Developers share such traces to discuss the debugging process and to communicate about problems. Since stack traces also provide significant insights on dynamic dependencies between entities, we take them into account (see Section 8.3.1). Stack trace lines, however, often exceed the maximum line length allowed in many email clients (*i.e.*, 80 characters), thus they may appear truncated (Figure 8.8).

```
at org.apache.jmeter.junit.JMeterTest.testGU
IComponents(JMeterTest.java:72)
```

Figure 8.8: Example interrupted stack trace line

By exploiting the regular structure of stack traces, we devised a regular expression for their normalization. The expression matches the candidate lines, even if scattered over many lines, and removes all unwanted line breaks.

Content splitting. Our fragment extraction phase must handle and filter a high number of ambiguities in the email text, generated by NL and incomplete and scattered fragments. Hence, an email with ample content requires considerable time to be parsed and might create scalability problems. Since our technique has to handle tens of thousands of documents in a reasonable time, we split the email in self-contained blocks to reduce ambiguities and the parsing time.

First, the splitting process divides the body in multiple parts according to the quotation levels: Each time we encounter a change in the quotation level we create a new split, preserving the correct order. Different quotation levels are already separate blocks, which respect the intention of the email's author, thus they are not disruptive with respect to the meaning of the parts. We analyze each split, and apply further splitting techniques when it is longer than a specified threshold (*i.e.*, 50 lines):

- **Code patches** Starting from the beginning of the split, when we encounter the header of a code patch, we create a new split. When referring to the same file, blocks of patches are independent, and are treated separately by the parser, with no information loss.
- **Stack trace lines** Starting from line 50 (to avoid the generation of too many splits), we split when we encounter a stack trace line. These lines can be analyzed separately without losing any contextual information.
- **Natural language lines** If we reach line 80 without having split before, we try to find lines with only NL, because they can be separated without breaking the structure of any code fragment. To recognize NL lines, we use the technique we previously devised (see Chapter 7), by which we can determine, with a considerable precision, whether a line is code. Since the method does not assure perfection, we split when it finds four consecutive non-code lines.
- **Forced splitting** If we reach line 150 without any split, we divide the content as soon as we encounter a line not containing an open parenthesis, or curly brackets. This forced splitting can alter the context, but, in practice, is applied to a very small fraction of emails.

8.4.2 Empirical Validation

To analyze the effectiveness of our extraction technique from two perspectives, we measure the capacity of our method in locating the chosen structured elements and assess whether the grammar production assigned to the fragments is correct. Table 8.9 details the results.

Figure 8.9: Systems' Mailing Lists and Results

System	Emails			Results	
	Population	with code	Sample	Precision	Recall
ArgoUML	24,876	12%	50	99%	95%
Freenet	22,095	9%	39	99%	97%
Mina	12,869	29%	99	99%	94%

In the second column, we report the mailing list size, after filtering out emails automatically generated (*e.g.*, by issue tracking systems).

We randomly chose a statistically significant sample of emails *with code* to inspect. Based on our previous work on classifying emails and lines containing source code (see Chapter 7), we know the proportion of messages with code in the chosen mailing lists (reported in the third column of Table 8.9). This allows us to use this information for calculating the size of significant samples [193]. The fourth column of Table 8.9 reports the number of emails from which we randomly selected samples. With these sample sizes, we are 95% confident that the emails represent the population with an error of 9%[1].

To evaluate the techniques to detect documents and lines containing source code fragments, we use two well known IR metrics presented in Section 4.2.4: *precision* (Equation 4.2), and *recall* (Equation 4.3).

Before manually creating a benchmark with all the emails in our samples, we processed emails with our normalization phase (Section 8.4.1). Then, we manually inspected each email and labeled all structured fragments with the correct grammar production. Conducting the normalization phase before the annotation had the beneficial side effect that it allowed us to verify the normalization process. Automating the comparison of the expected result (*i.e.*, the annotations in the emails) with the outcome of our approach can lead to problems in the evaluation itself. For example, a complete compilation unit in the benchmark could be recognized as a sequence of distinct pieces by our technique (*e.g.*, because it has been divided in multiple email splits). Classifying this behavior as completely erroneous would misreport the effectiveness. Hence, we manually inspected and compared the outcome of the approach to the benchmark.

The right-hand side of Table 8.9 shows the results obtained by the approach applied to the 183 emails containing code that we manually labeled. It achieved very high precision on the complete dataset for all three systems, with almost no NL words classified as parts of structured fragment. The recall shows that the method recognized almost all the required fragments. We found no error in the grammar productions in the recognized fragments.

Error Inspection. By manually inspecting the entire output of our technique, even though severely time-consuming, we gained a qualitative knowledge of the cause of the few errors generated. The majority of the false negatives (which affect the recall) were caused by the noisy nature of emails. In particular, a few code fragments were truncated in the middle of an identifier, thus hindering a correct complete identification (although the surroundings were recognized).

In Figure 8.10, the argument `url` of the method invocation `openProject` is truncated.

```
ActionOpenProject.getInst().openProject(ur
l);
```

Figure 8.10: Example truncated code fragment

Our approach only recognizes the first method invocation `ActionOpenProject.getInst()` (which has class scope), thus generating a false negative. This also underlines the role of manually evaluating the output: not considering the first method invocation as correctly recognized would be an error. Examples of false positives are the strings *"Java(TM)"* and *"developer(s)"*, which are wrongly reported as method invocations. Since such cases occur rarely, their impact on the model produced is insignificant. Still, they could be easily removed through post-processing, for example with statistical parsing.

1 This only validates the quality of sample sets as an exemplification of the populations, it is not related to the *precision* and *recall* values.

8.4.3 Threats to Validity

Threats to external validity. These threats concern the generalizability of our results. The approach we propose is built on a single object-oriented programming language, Java. Even though we relied on Java specific naming conventions (*e.g.*, camel casing) to disambiguate some constructs, similar, or even identical, conventions are used in a number of other widely spread programming languages, such as C#, Python, JavaScript. In addition, we introduced minor modifications in the standard Java grammar, thus we expect our approach to be easily adapted to other object-oriented languages with small effort.

Even though we evaluated the source extraction and model reconstruction phases on only three systems, we considered statistically significant sample sets and we achieved very promising results. In addition, the manual inspection of the results helped identifying future improvements.

Threats to construct validity. These threats regard measured variables that may not actually measure the conceptual variable. In our case, to assess the source fragment extraction phase, we relied on human judgment, both to label emails with the expected productions, and to evaluate the output. This process can be error-prone; to alleviate this, we did not directly evaluate the output on non-annotated emails, but we clearly separated the two phases. In the first one, we labeled emails without knowing the results of our approach. This allowed us to effectively verify all the expected source fragments. We decided to make use of human validation also for evaluating precision and recall. This choice is guided by the fact that the errors in an automated process would have been probably more significant than those of a human reviewer. Moreover, the human inspection allowed us to obtain a qualitative evaluation of our results.

Threats to statistical conclusion validity. These threats concern the relationship between the treatment and the outcome. In our approach for source code extraction evaluation, we considered samples to represent the population with an error of 9% at a confidence level of 95%.

8.5 iLander: Model Extraction

The model extraction process aims at analyzing the code fragments extracted in the previous phase to derive structural information about a system, thus supporting software understanding and development. The quality of the derived information depends on the number and nature of the available source code fragments. For instance, a method invocation like `URLDecoder.decode()` suggests the existence of both a class `URLDecoder` and a static method `decode` belonging to it. In contrast, a fragment with a complete compilation unit conveys more information on classes, methods, and relationships.

We implemented the model extraction phase using ASF-SDF. Our main extraction function takes as input a set of extracted code fragments and returns the set of *model facts* that can be derived. For example, Figure 8.11 shows the output of our approach applied to the second block (lines 4–6) in Listing 8.1.

For extracting basic model facts from each type of code fragment summarized in Table 8.1, we specified dedicated traversal functions that may call each other and may recursively call themselves. For example, if a `MethodDeclaration` includes some `MethodInvocation` s, the function that extracts facts from method declarations needs to invoke the function that extracts

```
 1 (ISLAND.Class
 2   (id: 3)
 3   (name 'URLDecoder')
 4   (startLine 3)
 5   (startColumn 0)
 6   (endLine 3)
 7   (endColumn 29))

 8 (ISLAND.Method
 9   (id: 4)
10   (belongsTo (idref: 3))
11   (name 'decode')
12   (hasClassScope true)
13   (startLine 3)
14   (startColumn 0)
15   (endLine 3)
16   (endColumn 29))
```

Figure 8.11: Example of extracted facts

model facts from method invocations. Similarly, a method invocation might consist of a chain of method invocations, hence the need for recursion.

Each extracted model fact receives a unique identifier in the form of an integer attribute (*id*, as in Figure 8.11, lines 2 and 9), which is recursively incremented during the parse tree traversal and is used to express links between separate model facts. Most model extraction functions have an *owner fact id* argument, which allows referring to a previously extracted model fact, when relevant (*e.g.,* line 10 in Figure 8.11 sets that the method belongs to the class).

To ensure traceability, each resulting model fact is linked to the fragment from which it was derived, using the exact code fragment location in the NL artifact (*e.g.,* lines 13–16).

8.5.1 Metamodel

We model the facts that we extract according to the FAMIX metamodel [181], a language independent metamodel of the static structure of object-oriented systems. It defines entities such as `NameSpace`, `Class`, `Method`, `Inheritance`, `Invocation`, `Parameter`, `Attribute`, *etc.* We extended FAMIX to handle the peculiarities of the fragmented information we are manipulating, and changed the prefix of the entities to `ISLAND` (*e.g.,* lines 1 and 8 in Figure 8.11).

The previous phase (Section 8.3) extracts a sequence of fragments from textual artifacts. Since each fragment is isolated from the others, any extracted structural information is partial, *i.e.,* the model fact extraction process results in *several* partial models, one for each source fragment, which need to be merged in a post-processing step. We identified heuristics that help to recover missing links, *e.g.,* it is likely that the same class corresponds to multiple class fact occurrences in the output. In this case, fact merging could be based on fact names (see the case study we present later). A more complex merge consists in exploiting the source fragment locations associated to different facts. For example, in Figure 8.1, line 4, the class name `ModuleLoader` and the method declaration `loadModulesFromExtensionDir` are in the same natural language sentence, thus one could formulate the hypothesis that the method belongs to the class.

8.5.2 Transformation Example

The whole ASF-SDF grammar and rules implemented for our approach are available, with examples, at: `miler.inf.usi.ch/iLander`. Here we show a descriptive example.

```
4  public void loadModulesFromDir(String dir) in ModuleLoader.java.
```

Figure 8.12: Incomplete method declaration from Figure 8.1

Let us consider the incomplete method declaration in Figure 8.1, line 4 (repeated in Figure 8.12); Figure 8.13 shows the output produced by the previous extraction phase. Incomplete method declarations contain the information on the existence of a method (*e.g.*, `loadModulesFromDir`), with additional optional classes specified by the formal parameters (*e.g.*, `String`).

```
( IMETHOD_DECLARATION :
  area-in-file("exampleDocument", area(4, 0, 4, 43, 148, 43)) :
  public void loadModulesFromDir(String dir) )
```

Figure 8.13: Extracted incomplete method declaration

We define a traversal rule (Listing 8.2) to model relevant facts from these declarations.

The parsing of each declaration appends three parts to the list of model facts already extracted from the same document. The first set of facts, `#Facts1`, is produced by the function `extractRT-Facts`, and returns the facts corresponding to the return type of the method (in our example, the declaration has no return type). `#FactID` corresponds to the `id` of the return type as generated by the function. The rule, then, produces a new `ISLAND.Method`, with the corresponding attributes, some of them produced by appropriate functions, such as `getModifierAttributes`, which generates access control attributes. The parameter `#OwnerID`, which is passed to the function, is propagated in the case of method declarations inside class declarations to create the ownership binding. The function `getBelongsToAttributeIfAny` creates this binding iff the passed value is strictly greater than `0`; calling functions pass the value `0` when the method declarations are isolated and thus the binding cannot be determined. The set of facts, `#Facts2`, is constructed by the call to the function `extractFacts` on formal parameters. This function call creates a new `ISLAND.FormalParam`, which models a formal parameter, together with the corresponding facts for the parameter type. Figure 8.14 details the final output of the model extraction phase, when applied to the incomplete method declaration in Figure 8.13.

Another interesting case comes from field declarations. For example, the fragment "*private MyClass field = new MyClass()*" contains three pieces of information: the presence of a field, a class, and a constructor invocation. Since the fragment is extracted as a field declaration, the rule in Listing 8.3 is applied.

```
extractFacts(#IncompleteMethodDeclaration,
             result(#Facts,#FactID),
             #Location)
= result (#Facts #Facts1
        (ISLAND.Method
          (id: #FactID1)
          (isConstructor false)
          (signature toSignature(#Identifier,#FParams))
          (name toString(#Identifier))
          getModifierAttributes(#Modifiers)
          (declaredType (idref: #FactID1))
          #Facts2,#FactID2)
 when #Modifiers #MethodRes #Identifier(#FParams)
                      := #IncompleteMethodDeclaration,
  result(#Facts1,#FactID1) :=
       extractRTFacts(#MethodRes,result( ,#FactID+1),#Location)
  result(#Facts2,#FactID2) :=
       extractFacts(#FParams,result( ,#FactID1+1),#Location,#FactID1,0)
```

Listing 8.2: ASF traversal rule for incomplete method declaration

The field declaration rule considers the structure as imposed by the language grammar and produces an `ISLAND.Attribute` fact, together with the corresponding facts coming from the type of the field. In this case, they would correspond to a class fact. Thus, we must analyze the right hand side of the assignment, that is, what the grammar specifies to be a `VarInitializer`. The traversal function would traverse the parse tree, and eventually reach the nonterminal corresponding to a `ConstructorInvocation`, and then extract the appropriate facts (Listing 8.4).

A generalization of the previous example would be a field declaration like *"private MyClass field = new OtherClass()"*. This fragment adds additional information, *i.e.*, the class `OtherClass` is assignable to `MyClass`. This could be the case of an inheritance relationship, according to FAMIX, but it is not the only case: the information is relatively loose. For example, we can neither determine if `MyClass` is an interface or a class; nor at which level of the inheritance hierarchy the two classes are located, that is, if there is a class `IntermediateClass` that extends `MyClass`, such that `OtherClass` extends it. More complex examples involve generic expression in field declaration assignments that would require a *type inference* mechanism.

8.6 iLander: Disclosing New Directions for Analyses

iLander creates one model for each document analyzed. This enables, for example, the reconstruction of the system model described in a design document, by processing the structured data it contains. In the output model, we would expect to find the most important components of a system, from an architectural perspective, and their relations. A comparison of this model with that obtained from the actual source code would spot inconsistencies or implementation issues not envisioned at design time, thus serving both developers and engineers. A work of document-to-classes traceability (see Chapter 4 would only connect the parts of the document

```
(ISLAND.Method
 (id: 5)
 (isConstructor false)
 (signature 'loadModulesFromDir(String)')
 (name 'loadModulesFromDir')
 (accessControlQualifier public) ... )

(ISLAND.Class
 (id: 6)
 (name 'String') ... )

(ISLAND.FormalParameter
 (id: 7)
 (name 'dir')
 (parentBehaviouralEntity (idref: 5))
 (declaredType (idref: 6))
 (position 0) ... )
```

Figure 8.14: Facts from an incomplete method declaration

with the corresponding (if any) entities in the system: It would not recover the system's "view" that a different model offers, it would not spot missing relationships, or it would not even inform about entities described in the documents that do not exist in the system. Only by recovering the alternative model, we can access this new information.

Similarly, when iLander is applied to a thread of emails (treated as a single document) discussing the same topic, it generates the system model as it emerges from the discussions and developers' view. We can thus analyze how developers envisioned a topic and get guidance to better comprehend the evolution of a system.

As a proof of concept, we apply iLander to the entire content of a mailing list pertaining to the development of a software system. Our purpose is showing how the information extracted offers a new perspective on the system, which can be used for software reconstruction and analysis.

We focus on ArgoUML (see Section 3.4), since its developers have been using the public mailing list as their main form of communication for nearly 10 years. Hence, we expect it to contain relevant structured information about the system. We use our approach on the complete mailing list from its inception (January 2000) to the start of 2010.

8.6.1 Model reconstruction

By applying iLander to a stream of documents, we obtain multiple models of the system that is discussed: One for each document analyzed. In fact, our approach treats each document separately and produces facts that are not connected to those generated from other documents. For example, in the case of the chosen mailing list, although we might find data about the same entities of the system multiple times across the emails (*e.g.*, the class `ProjectBrowser` appears in 306 emails), the models are independent.

```
extractFacts(#FieldDeclaration,
                    result(#Facts,#FactID),
                    #Location, #OwnerID)
= result (#Facts
        (ISLAND.Class
          (id: #FactID)
          (name toString(#Identifier1))
          getLocationAttributes(#Location))
        (ISLAND.Attribute
          (id: #FactID + 1)
          (name toString(#Identifier2))
          (declaredType (idref: #FactID))
          getModifierAttributes(#Modifiers)
          getBelongsToAttributeIfAny(#OwnerID)
          getLocationAttributes(#Location))
        #Facts2, #NewFactID)
 when #Modifiers #Identifier1 #Identifier2 = #VarInitializer; :=
    #FieldDeclaration,
  result (#Facts2, #NewFactID) :=
       extractFacts(#VarInitializer, result( ,#FactID+2),#Location)
```

Listing 8.3: ASF traversal rule for `FieldDeclaration`

For this reason, the first step is to obtain a *single* and coherent model of the system. We achieve that by following the *temporal stream* of the emails and by adding step-by-step the modeled information, from the past to the present. For example, first, we might discover the presence of a class `A`, then, we could learn that it belongs to the namespace `z.y`, along with other classes. Then, in more recent emails, we may find the methods declared in `A`. In the case of inconsistent information (*e.g.*, first we find that the class `A` extends the class `B`, then that it extends the class `C`), we drop the older data and replace it with the new knowledge. With this approach, we gradually build a coherent and unified model.

Once we obtain the unified model, we assess its consistency with respect to a model of the system extracted from its source code[2], and we measure the percentage of information about it that the mailing list provides.

We compare the models at the level of classes, since the class is the fundamental building block of object-oriented systems. We consider that we have information about a class when we find at least one email with a structured fragment related to it. For many of the classes, our approach extracts and reconstructs methods of classes, invocations, attributes, and more structural information (*e.g.*, relationships between classes), but, for the sake of simplicity, in this validation we focus on the *existence* of classes in the mailing list.

Table 8.2 reports the result we obtained in the model reconstruction. The first column contains the number of classes in the system, the second reports how many of these classes we also extract from the mailing list. The third column expresses the percentage of classes extracted. The last

2 We consider ArgoUML 0.29.4, the first release in 2010.

```
extractFacts(#ConstructorInvocation,
                    result(#Facts,#FactID),
                    #Location)
= result (#Facts
        (ISLAND.Class
          (id: #FactID)
          (name toString(#Identifier1))
          getLocationAttributes(#Location))
        (ISLAND.Method
          (id: #FactID + 1)
          (name toString(#Identifier2))
          (isConstructor true)
          (belongsTo (idref: #Integer))
          getLocationAttributes(#Location))
        #Facts2, #NewFactID)
 when new #Identifier1 (#MethodArgs) := #ConstructorInvocation,
  result (#Facts2, #NewFactID) :=
   extractFactsFromArgs(#MethodArgs, result(,#FactID + 2), #Location,0),
     isAClassName(#Identifier1) == true
```

Listing 8.4: ASF traversal rule for `ConstructorInvocation`

column is the minimum number of developers who must have worked on a class for counting the class itself.

Table 8.2: Percentage of system reconstruction from emails

Classes			Developers	Classes			Developers
in system	**in emails**	**ratio**		**in system**	**in emails**	**ratio**	
1,853	823	44%	1	339	289	85%	9
1,818	819	45%	2	239	212	89%	10
1,548	781	50%	3	186	169	91%	11
1,252	684	55%	4	138	127	92%	12
956	565	59%	5	94	87	93%	13
688	478	69%	6	63	61	97%	14
559	417	75%	7	41	41	100%	15
439	353	80%	8				

The first row shows that we find information about 44% of the system entities simply by processing the content of its mailing list. Even though we consider this to be a good result, we note that more than half of the entities are never considered in the structured fragments of discussions.

We postulated two hypotheses for interpreting this observation: 1. the code is highly modularized, thus only entities exposing behavior to other modules are discussed in the mailing list, and 2. entities which are under the responsibility of a single or few developers are less likely to be discussed with the whole community.

To explore the first hypothesis, we manually inspected the modules of the system and we found a recurring pattern. As an example, we consider the module that manages the importing of C# code into ArgoUML. This module contains 114 classes, but only two of them appear in the mailing list: `CSModeller` and `Parser`. The latter is the core of the implementation of the module, while the former is the only class that exposes methods called by other modules. Another example is the module that manages the deployment of the UML diagram in the user interface. This module contains 19 classes, with only two reconstructed from the mailing list. Similarly to the previous case, these classes, `FigObject` and `UMLDeploymentDiagram`, are respectively the interface for the other modules, and the core class of the module itself. The same pattern occurs in other modules and packages corroborating our first hypothesis.

To investigate the second hypothesis, we considered classes of the system that have been developed by more than `n` developers in their history (the values of `n` are in the column `Developers` of Table 8.2). The data shows that the more people worked on a class, the more likely our technique recovers the class from the emails. This supports our hypothesis that classes with a few authors are less discussed.

These findings open up questions: Is it appropriate that some parts of the system, being under the responsibility of a few developers, are not discussed with the community? What would be the impact in case such developers leave the project? Our technique is able to reveal that a relevant part of the system is never discussed among developers on the official channel. Should they be discussed more, or documented externally?

Another question takes into account classes that can be reconstructed from the mailing list: Are these classes the most important from a structural perspective, since they involve more developers and are connecting points among modules? If so, are these classes a good entry point to be studied by a new developer joining the project? These are questions that might trigger future research starting from these new analyses.

8.6.2 System analysis

On the one side, as just presented, our approach allows us to reconstruct a model of the system that is as close as possible to the one extracted found in the source code. On the other side, a researcher can use iLander to build *new perspectives* on the system. We suggest here one approach taking this direction.

Figure 8.15 shows a graph where the classes extracted from the mailing list are the nodes. The size of the nodes is proportional to the number of emails from which we could recover these classes, and the color denotes the date of the last email with a reference (darker nodes represent classes referenced more recently). The edges connect classes that appear together in at least `k` emails (for readability, we only show nodes that have at least one connection). In our case study, we found that `k` values between 15 and 25 show interesting clustering behaviors.

The graph puts in evidence that some classes are very popular in the mailing list. These classes are also among those mentioned most recently, and form cohesive clusters. By inspecting the graph node by node, we find a cluster formed by Java library classes that are significant for the system, such as trend management classes (used for the visualization of the UML diagrams) and exception classes. Other classes are also put in evidence by the graph, because of their size and connections. These classes can be good candidates for program analysis. We use our data to perform an analysis of the trend of their popularity in the mailing list, in order to gain in-depth insights about which emails can be the most significant to read.

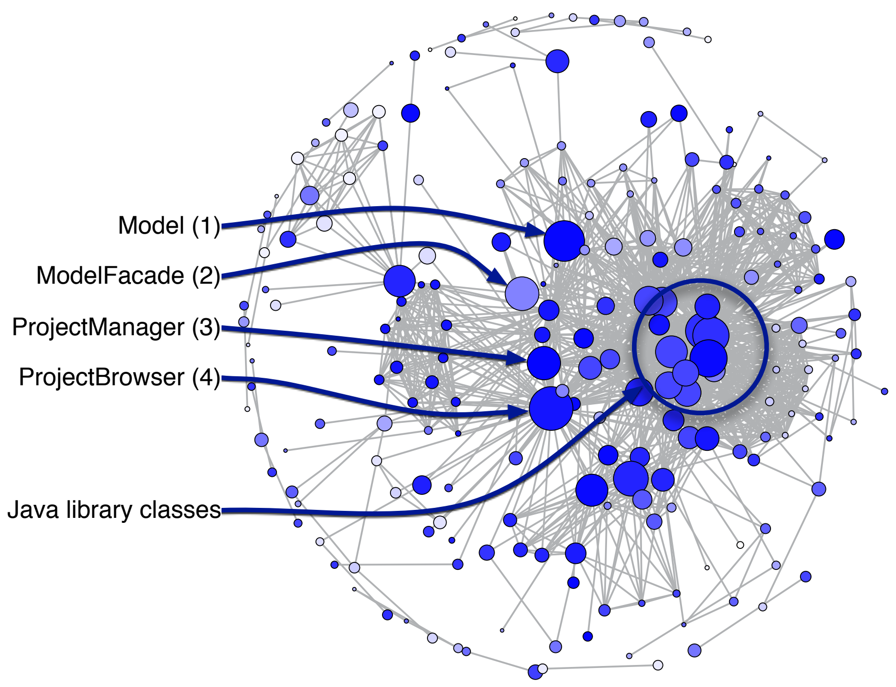


Figure 8.15: Class fragments popularity and relationships in emails

We divide the overall time interval of the mailing list into equally spaced time bins (by considering the amount of emails exchanged in the ARGOUML mailing list, we divide in bins of 90 days, which is a good trade off between precision in finding trends, and the number of emails to read in an interesting period), then we calculate, in each time interval, the number of distinct messages with fragments related to the chosen classes, and we assign it to the time bin. Finally, we normalize the value for each time bin (by the maximum number of emails referencing the class), to have a value between 0 and 1. Figure 8.16 shows the result. The area is colored according to the overall popularity of the classes under analysis.

Figure 8.16 shows that all the classes have a single period in which they gained high popularity. Our hypothesis is that knowing the period in which a *critical event* in the life of a class occurs can help to understand important aspects of the evolution and the design of a system. With this trend analysis we find such moments in time. In order to investigate our hypothesis, we read all the emails in the "hot" periods.

In the emails related to the `Model` class in the second trimester of 2005, we discover that the class was at the center of a conspicuous refactoring: The developers decided to change the metamodel they have been using for modeling UML diagrams. That decision impacted the rest of the history of the system.

ModelFacade was also used in the implementation of the previous metamodel, and was transformed in multiple interfaces after a major refactoring and deprecated with the move to the new metamodel. This occurred after the period of major popularity of the class, in which developers discussed about how to improve it and, then, to eventually deprecate it.

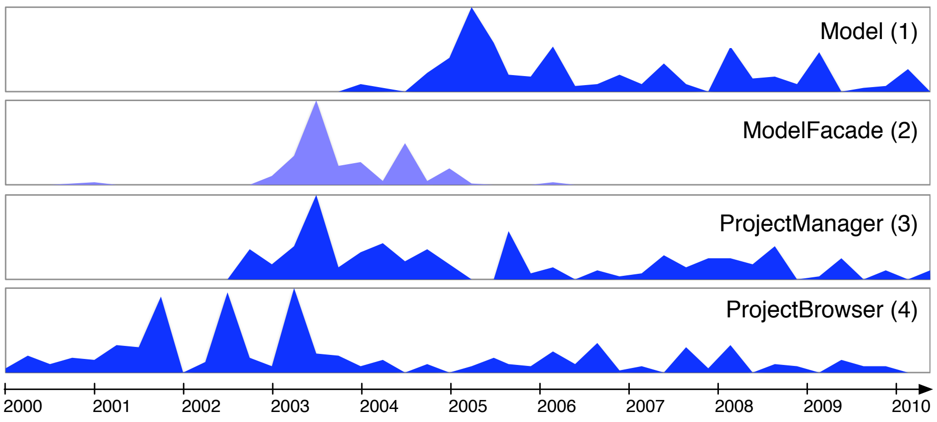


Figure 8.16: Trends in popularity of fragments of classes

We discover that `ProjectManager` is a class still used in ArgoUML for handling the project of the user. The emails in the peak of popularity are related to periods in which it was causing failures due to its defective loading functionality, and then fixed. By reading emails in the second peak (late 2005), we learn that `ProjectManager` was also involved in a metamodel change.

Analyzing the emails in the most relevant period for `ProjectBrowser`, we discover that in 2003 the class "*is used everywhere in the argo code*", and developers wanted to migrate part of its responsibilities to other classes. By looking at the whole subsequent releases of the code, we confirmed how the amount of entities in the system referencing this class drops significantly, while it still remains present in the system until the most recent versions (as spotted also by the continuous messages referencing it).

8.7 PetitIsland: The Parsing Approach

In the previous sections, we showed our iLander approach, which uses GLR parsing to implement island parsing using a scannerless approach based on the SDF grammar definition formalism. Due to the time complexity, our preliminary results were not satisfactory in terms of performance. Moreover, we found it difficult to evolve and maintain it, due to its underlying programming environment. For these reasons, we adopted a different approach to island parsing that avoids dealing with ambiguities with context-free grammars and provides better performances. Moreover, we managed to implement our approach into our Miler toolset and devise it as a framework that we named PetitIsland.

PetitIsland exploits different parsing methodologies while combining the best properties of each of them, in order to achieve:

- *accuracy* and *efficiency*, by using parsing expression grammars (PEGs) [74], that enable precise island parsing in linear time;
- *flexibility* with scannerless parsers, enabling the disambiguation between different alternatives using lexical properties of tokens, like letter case;

- *extensibility*, realized with parser combinators, that enable easy and modular description of different structured fragments embedded in arbitrary text.

We implemented PetitIsland using the parser framework PetitParser [163], which supports all the parsing techniques described above.

8.7.1 Parsing Expression Grammars

PetitIsland is based on Parsing Expression Grammars (PEGs). Instead of expressing a language in a generative manner, like context-free grammars, PEGs use avoid expressing equivalent non-deterministic choices between alternative productions. They use an *ordered* choice operator, usually denoted with `/`. Such an operator lists each alternative in a prioritized order; thus, in a PEG, the first of the alternatives that successfully matches is chosen. For example, the following PEG parses arithmetic expressions with sums and products, giving priority first to product, then to sum; thus, it produces the parse tree on the right of Figure 8.2.

```
Term ← Prod + Term / Prod
Prod ←  x * Prod / x
```

PEGs are a powerful formalism that is essentially recognition-oriented; a PEG can be considered as a formal description of a top-down parser. However, PEGs have more expressiveness of typical classes of grammars associated with top-down parsers, and they can also recognize non-context-free languages. Because of their recognition-oriented nature, and the use of ordered choice, PEGs parse languages in linear time [74].

8.7.2 Implementation

We implemented our island parsing approach, PetitIsland, in PetitParser, a parser framework to model grammars and parsers with scannerless parsing, parser combinators, and parsing expression grammars [163]. We adopted the Smalltalk version of PetitParser, which has a convenient syntax[3].

PetitParser in a Nutshell

PetitParser grammars are specified by means of primitive parser objects (see Table 8.3) that are composed into other parsers using parser combinators (see Table 8.4[4]).

A PetitParser parser for an identifier, for example in the form of a letter followed by zero or more letters or digits, can be implemented as follows:

```
identifier := #letter asParser ,
        (#letter asParser / #digit asParser) star.
```

3 `http://scg.unibe.ch/research/helvetia/petitparser`

4 The entire list can be found in `www.lukas-renggli.ch/blog/petitparser-1`.

Table 8.3: A Selection of The Parser Combinators in PetitParser

Terminal Parser	Description
$a asParser	Parses the character ‘a’.
abc’ asParser	Parses the string ‘abc’.
#any asParser	Parses any character.
#digit asParser	Parses the digits ‘0..9’.
#letter asParser	Parses the letters ‘a..z’ and ‘A..Z’.
#uppercase asParser	Parses the letters ‘A..Z’.

Table 8.4: A Selection of Terminal Parsers in PetitParser

Parser Combinator	Description
p1 , p2	Parses ‘p1’ followed by ‘p2’.
p1 / p2	Parses ‘p1’, if that fails parses ‘p2’.
p star	Parses zero or more ‘p’.
p plus	Parses one or more ‘p’.
p end	Parses ‘p’ and succeeds at the input end.

The expressions `#letter asParser` and `#digit asParser` return parsers that accept a single character of the respective character class; the `,` operator combines two parsers into a sequence parser; the `/` operator combines two parsers into an *ordered choice* parser, and the `star` operator accepts zero or more instances of this *ordered choice* parser.

By subclassing the PetitParser class that implements composite parsers (*i.e.*, `PPCompositeParser`), grammars can be defined as parts of a class. Each production is implemented with an instance variable and a method returning the grammar of the rule. For example, we can re-define our parser for identifiers more verbosely with the following class `PPIdentifier`:[5]

```
PPCompositeParser subclass: #PPIdentifier
  instanceVariables: 'validCharacters'
```

Beginning with the mandatory method `start`, that specifies the starting production, the class `PPIdentifier` declares the following methods to define the grammar:

```
PPIdentifier>>start
  ^#letter asParser , validCharacters

PPIdentifier>>validCharacters
  ^(#letter asParser / #digit asParser) star
```

We expanded the identifier production in two productions to highlight how instance variables and methods are used. The result is a parser made of a graph of connected parser objects, which can be used to parse input text:

5 See `www.lukas-renggli.ch/blog/petitparser-2` for more detailed examples.

```
parser := PPIdentifier new.
parser parse: `ex1'.      This returns an abstract syntax tree.
parser parse: `2ex'.      This returns a parse failure.
```

Island Parsing with PetitIsland

Thanks to parsing combinators and PEGs, we can define the basis of our entire island parsing approach in four productions. By using PetitParser, we implemented this in `PPIsland`:

```
PPCompositeParser subclass: #PPIsland
  instanceVariables: 'island water waterBlob'
```

We define the first production in the `start` method:

```
PPIsland>>start
  ^(island / water) plus end
```

The `start` production builds on the ordered choice provided by PEG: We specify that the `island` production has precedence over the `water` one (*i.e.*, `island / water`). Moreover, we set (using `plus`) that the text might contain one or more occurrences of `island` and/or `water` and must be parsed to the end (using `end`).

The second production we define is the `island:` method:

```
PPIsland>>island: aParser
  island := aParser
```

In this method, the `island` production must be declared externally and passed to the `PPIsland` parser as an argument (this is usually done when the `PPIsland` class is instantiated). Thanks to parsing combinators, we can leave the definition of island(s) external and have an approach to island parsing that is reusable out-of-the-box, under any definition of `island`.

The third and fourth productions regard the water:

```
PPIsland>>water
  ^waterBlob / #any asParser

PPIsland>>waterBlob
  ^(#letter asParser / #digit asParser) plus
```

The most conservative solution to define `water` would use the `#any asParser` expression, which consumes one single character of any kind (see Table 8.3). In practice, with this approach, the `start` production first tries to match an `island`, then, in case of failure, it matches and consumes any character (*i.e.*, `water`, defined as `#any asParser`), and it starts again from the subsequent character. Structured fragments do not start in the middle of a word, so we also defined `waterBlob`: A production that consumes an entire word instead of a single character (also speeding up the parsing).

To clarify the functioning of our island parsing approach, we present an example in which we plug a small parser into it. Let us assume we want to create a parser to recognize, within an arbitrary text, all the occurrences of words starting with an uppercase letter. First, we write a parser for the content we want to extract (*i.e.*, words starting with uppercase letters):

```
wUp := #uppercase asParser, #letter asParser star
```

We plug this parser into a new instance of `PPIsland`, creating a customized island parser, to parse an example text:

```
islandParser := PPIsland newWithIsland: wUp.
islandParser parse: 'an Example tExt.'.
```

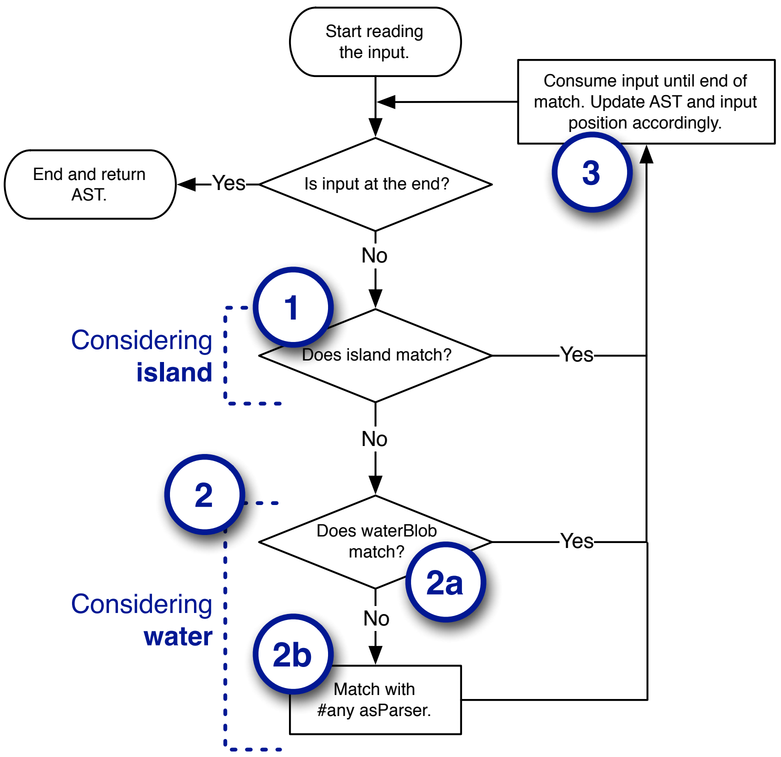


Figure 8.17: The parsing process, showing the role of precedences.

Figure 8.17 depicts the parsing process, underlining the flow and precedences of the parsers. Figure 8.18 depicts the final result of the parsing.

The `islandParser` starts from position `1` (Figure 8.18) containing the first character `a`. According to the `start` production, which specifies the main precedence, `islandParser` first tries to find a match with the `island` parser (Point 1, Figure 8.17). In this case, the `island` parser corresponds to the plugged `wUp` parser. Since `wUp` fails, the `islandParser` does not consume the input and rolls back to the `water` parser (Point 2). The `water` parser, in turn, first tries to match with `waterBlob` (Point 2a). From position `1`, `waterBlob` successfully matches (and consumes the input, Point 3) up to position `2` included (Figure 8.18); it also updates the resulting abstract syntax tree (AST) accordingly. After this, the `islandParser` continues from position `3` trying to match other

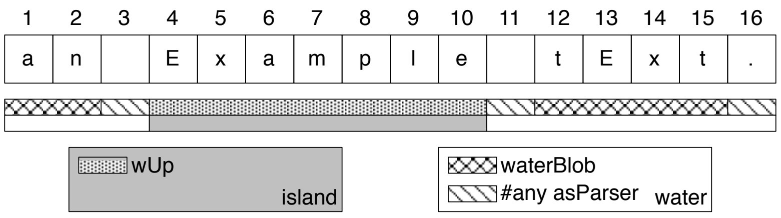


Figure 8.18: Parts matched with a small island parser.

islands and water (due to the `plus` in the `start` rule), always giving precedence to the former. From position `3` the only parser that matches is the `#any asParser` (thus reaching Point 2b in Figure 8.17), so a single character is consumed. From position `4`, the `wUp` parser matches the input and consumes it up to position `11`. The process goes on similarly until it reaches the end of input (due to the `end` in the `start` rule).

By using a transformation or subclassing `PPIsland`, it is possible to ignore water and return an AST with only islands.

Islands with Lakes: In documents, such as emails or website pages, that do not have to comply with a formal grammar, structured fragments often embed parts that are extraneous to their grammar. Figure 8.19 shows a classical example: Lines 2–6 contain a method declaration in which parts of the content are omitted and replaced with ellipses (lines 3 and 6).

```
1  Here you find a code example to answer your question:
2   public void setEnclosingFig(Fig each) {
3    ...
4    if (each == null || (each.getOwner() instanceof MPackage)) {
5     m = (MNamespace) each.getOwner(); }
6    ... }
7  The key point is the condition. [...]
```

Figure 8.19: Structured fragment embedding water and islands: Island with lakes.

These fragments are named *islands with lakes* [136], because are islands that embed water. We implemented a class `PPIslandWithLakes`, subclassing `PPIsland` and overriding the `start` method, to support these types of constructs:

```
PPIsland subclass: #PPIslandWithLakes
  instanceVariables: 'startParser stopParser'

PPIsland>>start
  ^startParser ,
     (island / (stopParser not , water) ) star ,
   stopParser
```

Differently from `PPIsland`, this parser also requires a `startParser` and a `stopParser`, which define the boundaries. Any definition of island can be plugged also in this case. After the starting boundary is found (*i.e.*, `startParsers` matches), the parser tries to match zero or more islands and lakes (*i.e.*, `(island / (stopParser not , water) ) star`). When no island is matched, `PPIslandWithLakes` tries to match with `water`, but only in the case it does not find a `stopParser` (*i.e.*, `stopParser not , water`). Whenever it finds something matched by the `stopParser`, `PPIslandWithLakes` stops.

To clarify the functioning of our island with lake parsing approach, we show how we can define a parser for recognizing the structured fragment in Figure 8.19. If we already defined a parser able to recognize the header of a method declaration (*e.g.*, the one in line `2` in Figure 8.19), we would define the parser for the method declaration body, and combine them:

```
methodDeclHeader := [ definition of parser ]
methodDeclBody := PPIslandWithLake
    newWithIsland: blockStatement
    start: ${ asParser
    stop: $} asParser.
methodDecl := methodDeclHeader , methodDeclBody.
```

We embed the `methodDecl` parser into a new instance of island parser (as previously done with the `wUp` parser):

```
islandParser := PPIsland newWithIsland: methodDecl.
```

This `islandParser` is able to correctly parse the example in Figure 8.19, thus extracting the method declaration and its non-water content.

8.7.3 Summary

In this section we detailed our approach to island parsing as it is implemented in PetitParser. We showed the importance of the *ordered choice* provided by PEG and how we use the flexibility granted by parser combinators. Using these concepts we created PetitIsland, a framework that is simple, yet powerful enough to define island grammars for various needs and seamlessly combine them. In the following section, we validate our approach by defining a complex island grammar to parse source code fragments written in Java, building on top of our PetitIsland framework.

8.8 PetitIsland: Validation

To validate the effectiveness of our approach to island parsing, we use it to extract and parse Java source code fragments embedded in NL text. Given the complexity, ambiguous nature, and size of a programming language grammar, we consider extracting source code fragments to be a veritable acid test to validate our approach to island parsing.

We started by implementing into PP the entire latest specification of the official Java language [83]. In practice, we created a class `PPJavaSyntax` by extending the PP class `PPCompositeParser` and implemented each production in the language specification with an instance variable and a method returning the grammar of the production (see Section 8.7.2 for details). For explanatory purposes, the authors of the language specification described many productions with left-recursive rules, as in the following:

```
TypeDeclarations:
  TypeDeclarations , TypeDeclaration
  TypeDeclaration
```

Since left-recursive rules cannot be directly implemented into PEGs [74], we re-wrote all these rules to avoid left-recursion.

8.8.1 Productions

The JAVA grammar has a starting production, which we implemented in `PPJavaSyntax` through the following method:

```
PPJavaSyntax>>compilationUnit
  ^(annotations optional , packageDeclaration) optional , importDeclaration
     star , typeDeclaration plus
```

This can be plugged into an instance of `PPIsland` by defining it as the `island` (see Section 8.7.2). In this way, however, only this production would be recognized within NL documents, while many others that often appear (*e.g.*, method invocations or declarations, if or do statements) would be lost. For this reason, we defined a catalogue of productions, listed in Table 8.5, that we want to recognize regardless of their surrounding context of NL sentences.

Table 8.5: Considered Java productions, in priority order (top to down, then left to right)

Nonterminal	Description
compilationUnit	Class declaration with package imports
packageDeclaration	Package declaration
typeDeclaration	Class or interface declaration
methodDeclaration	Method declaration
incompleteTypeDeclaration	Incomplete class or interface declaration
incompleteMethodDeclaration	Incomplete method declaration
strictFieldDeclaration	Strict field declaration
creatorWithOptSemicolon	Constructor invocation with optional semicolon
assertStatement	Assertion predicate
ifStatement switchStatement	Conditional blocks
forStatement whileStatement doStatement	Loops
breakStatement continueStatement	Execution statements
tryStatement throwStatement	Exception statements
synchronizedStatement	Mutual exclusion statements
returnStatement	Method return statement
classRelationship	Implements or extends relations
strictVariableDeclaration	Strict variable declaration
strictExpressionStatement	Strict expression statement
strictMethodInvocation	Strict method invocation
strictAnnotation	Strict annotation

Exploiting the PEG ordered choice, we ordered the considered productions from the most comprehensive down, before plugging them into a new instance of `PPIsland`. In this way, we do not lose the binding among the parts in case of larger productions. For example, consider the fragment in Figure 8.20.

```
This is the class implementing the Fibonacci algorithm:
 package com.stackoverflow;
 import java.io.*;
 class Fibonacci {
 [omitted code]
 }
Please note that this is a recursive solution.
```

Figure 8.20: Example text with a compilation unit.

In this case, by first trying to match a `compilationUnit` (which also includes the optional `packageDeclaration`), we realize that the `Fibonacci` class is defined in the `com.stackoverflow` package. If we tried to match first the single `packageDeclaration`, we would have lost the connection between the package and the class declaration.

Some of the considered grammar productions (Table 8.5) are translated to our approach from the JAVA language specification. This is the case of conditional blocks (*e.g.*, `ifStatement`), loops, and execution or exception statements. Other productions, described in the following, are derived or inspired from the original ones specifying correct JAVA syntax and from other programming customs, like naming conventions. Such new productions are needed to support island parsing and are a source of differentiation from traditional programming language grammars; they are needed, for example, to parse incomplete fragments or island with lakes. We implemented an island parsing JAVA PEG grammar in a new class: `PPJavaIsland`, which subclasses `PPJavaSyntax`. In this way, we only had to implement changed and new productions.

We defined the considered productions, shown in Table 8.5, in the new method `islands`:

```
PPJavaIsland>>islands
  ^(compilationUnit / packageDeclaration / importDeclaration / [... continues
    with all the productions in Table 8.5, in order.])
```

Then, we defined the island parser by plugging the parser for the consider productions into a new `PPIsland` instance:

```
javaProductions := PPJavaIsland new.
islandParser :=
  PPIsland newWithIsland: (javaProductions islands)
```

The derived productions enable the parsing of the following specific island parsing structures:

- *incomplete productions*, w.r.t. the original JAVA language productions;
- productions respecting *naming conventions*, that help disambiguate some specific JAVA fragments that may appear in NL sentences;
- *additional productions*, and *strict productions* that enable, for example, parsing of class relationships appearing in the text;
- *island with lakes*, that is, islands containing embedded NL text (i.e., water, hence a *lake*).

Incomplete Productions. Incomplete productions are the first source of differentiation from the standard grammar. As an example, we consider lines 2–3 in Figure 8.21.

```
 1 [...] how I solved it. WallSetter implements AsyncTask, you
 2 should not forget to implement the method protected void
 3 onPostExecute(String result).
 4 Your other class, instead, should look like this:
 5 public class Square extends Shape {
 6  private length = 5;
 7  public Square(){...}
 8  ...
 9  public double area(){ return length * length; }
10 }
11 So that your last class can call the method area() to get it.
```

Figure 8.21: Example text with various source code fragments.

The document author is explicitly referencing a method signature, but the body is missing. The standard Java grammar includes only method definitions that are followed by either a semicolon (when they appear in interfaces) or the whole method body. By considering only the fragments that can be reduced to a nonterminal in the standard grammar, we might lose several structured fragments, such as the one in the example above. For this reason, we also extract fragments corresponding to a subset of a production that does not reduce to a nonterminal in the standard language specification.

Although every possible incomplete production can be supported, for this validation we limit ourselves to the most popular incomplete productions found in NL documents: incomplete method and type declarations. Such incomplete productions do not require a final semicolon or a block with the body of the construct. For example, we implemented the following production in `PPJavaIsland` to support incomplete method declarations (as the one in Figure 8.21):

```
PPJavaIsland>>incompleteMethodDeclaration
  ^methodModifiers optional, typeParameters optional , (voidType / ncrType)
     optional, identifier , ncrStrictFormalParameters ,
    emptySquaredParenthesis star , throws optional
```

Naming Conventions Respectful Productions. The aforementioned `incompleteMethodDeclaration` method includes nonterminals that start with the prefix `ncr` and are extraneous to the official grammar (*e.g.*, `ncrType`). The acronym `ncr` stands for *naming convention respectful*; the corresponding productions are included to reduce some of the ambiguities that might arise from parsing Java fragments in NL documents, by exploiting Java naming conventions [184].

Consider, for example, the fragment *"method area()"* (Figure 8.21, line 11). By using the standard `type` instead of `ncrType`, the `incompleteMethodDeclaration` parser would match *"method area()"* as a valid fragment: *"method"* is a valid identifier, and thus a lexically valid return type for the Java grammar. However, Java naming conventions prescribe that class names must be capitalized, thus *"method"* violates the naming conventions. The name `ncrType` indicates that we implemented the production in a way that only a type that respects the naming conventions will be accepted. We exploit this to exclude the reductions where the supposed types are likely to appear by chance and thus be invalid.

Additional Productions. Class relationships are another kind of structured fragment containing interesting information to be extracted. They express inheritance and implementation relations between classes and interfaces. For example, in line 1 of Figure 8.21, we find two potential class names separated by the keyword `implements`. From this fragment, we could derive that `AsyncTask` is an interface, and there is an implementation relation between the two entities.

These fragments can be recognized as follows:

```
PPJavaIsland>>classRelationship
  ^ncrStrictIdentifier, (extends / implements), ncrStrictIdentifier
```

Strict Productions. We also defined some productions whose name starts with, or includes, the prefix *strict* (*e.g.*, `strictAnnotation` or `ncrStrictIdentifier`). In these cases, we exploit the scannerless parsing approach (Section 8.2.1) for disambiguation.

Consider, for example, the fragment "*I solved it. WallSetter implements AsyncTask*" (line 1 in Figure 8.21). By using a standard `identifier` (or even a `ncrIdentifier`) the `classRelationship` parser (see Section 8.8.1) would match: "*it. WallSetter implements AsyncTask*", thus recognizing `it.  WallSetter` as an identifier. In fact, the Java grammar allows identifier with qualifiers separated by spaces. However, this is not recommended by the naming conventions. Since we do not have a separate tokenization phase, as we rely on scannerless parsing, we can specify that certain productions require a more "strict" tokenization, by not allowing whitespace between certain parts. For example, `ncrStrictIdentifier` does not allow whitespace in a type name, and `strictAnnotation` does not allow whitespace between the `@` and the identifier.

Productions with Lakes. Some of the considered productions, when appearing in arbitrary documents, might contain embedded water. This mainly affects productions with a body, such as the one in declarations and loops, that includes multiple statements. Figure 8.21 shows an example of a class declaration (lines 5–9) and a method declaration (line 8) that contain water. To correctly support these cases, in `PPJavaIsland`, we overrode the methods defining body productions; for example, this is how the production for a `block` (used, for example, by loops) looks like to support embedded water:

```
PPJavaIsland>>block
  ^PPIslandWithLake
    newWithIsland: blockStatement
    start: ${ asParser
    stop: $} asParser.
```

8.8.2 Results

As a case study to validate our approach to recognize Java code fragments in NL artifacts, we focus on Stack Overflow[6]. Stack Overflow is a web service providing developers with the infrastructure to exchange knowledge in the form of questions and answers: Developers ask questions and receive answers regarding issues from people that are usually not part of the same project. Post authors can include fragments of source code and tag them, so that they appear formatted in the most appropriate way, with text highlighting.

The advantage of using Stack Overflow data is that it is already code tagged, by external people not involved with the evaluation, and that it corresponds to a real-world scenario for applying our approach. The sample of questions considered in the benchmark is the same used by Rigby and Robillard [167]. It is composed of 188 answers, taken from posts with project tags related to three Java applications: HttpClient[7], Hibernate[8], and Android[9]. The three project tags crosscut a diverse set of topics. We downloaded the dataset kindly provided by Rigby and Robillard[10] and adapted it to our task. In fact, some of the answers where not correctly code tagged by their authors: Sometimes the `code` tag was used to format differently a certain piece of text that was not real source code. We manually re-inspected all the 188 answers and fixed incorrect code tags. The final benchmark is publicly available.

With the benchmark in place, we applied our approach to island parsing to the raw text and wrote a script to automatically compare what we extracted to what was tagged as `code` in the benchmark. To quantify the effectiveness of our approach, we use two well known IR metrics presented in Section 4.2.4: *precision* (Equation 4.2), and *recall* (Equation 4.3). Table 8.6 describes the datasets and the results obtained, by project tag. Our approach reached an average precision of 98% and recall of 96%. This means that in all three project tags, the approach recognized the vast majority of the required fragments, almost with no errors.

Table 8.6: Dataset description and results, by project tag

Project Tag	**Sample**		**Results**			
	Answers	**Code Fragments**	**FP**	**FN**	**Precision**	**Recall**
Android	63	120	2	5	98%	96%
Freenet	51	68	1	2	98%	97%
Mina	74	163	3	6	98%	96%

The automatic comparison we set up is very strict. Consider the code fragment in Figure 8.20 (repeated in Figure 8.22).

The expected outcome is a single code fragment, which corresponds to a compilation unit. If our approach did not recognize this as a single piece, but as two or more pieces (*e.g.*, a `packageDeclaration`, followed by an `importDeclaration`, plus a `typeDeclaration`), we would have

6 `http://stackoverflow.com/`
7 `http://hc.apache.org`
8 `http://www.hibernate.org`
9 `http://developer.android.com/about/index.html`
10 `http://swevo.cs.mcgill.ca/icse2013rr`

```
package com.stackoverflow;
import java.io.*;
class Fibonacci {
[omitted code]
}
```

Figure 8.22: Source code from Figure 8.20.

counted one false negative and as many false positives as the separated fragments proposed (*e.g.*, three). Also in the case of partial extractions, we count an incomplete fragment as both a false positive and a false negative. For example, if we extracted `onPostExecute(String result)` from Figure 8.21 (repeated in Figure 8.23 instead of `protected void onPostExecute(String result)`, we would have counted a false negative (for the missed fragment) plus a false positive (for the partially wrong extraction).

```
should not forget to implement the method protected void
onPostExecute(String result).
```

Figure 8.23: Example lines with code from Figure 8.21.

Error Inspection. We manually inspected the errors generated by our approach to understand their causes and whether they could be addressed. Most errors were due to ambiguities that cannot be resolved without a deeper understanding of the meaning of the text. For example, in the sentence "*A new openConnection() method has been added*", our parser recognized a constructor invocation: `new openConnection()`. This error could only be avoided knowing that `new` was part of the discourse, rather than a valid fragment of code. Fixing this error with a lexical parsing approach like ours is possible by either excluding similar cases from the available productions or devising stricter rules for recognizing them. Both these solutions would fix such a false positive, but they could introduce new false negatives, with the final result of only rebalancing the trade-off between precision and recall.

8.8.3 Summary

In this section, we validated our approach to island parsing by extracting source code fragments from NL documents. This task is useful for a number of applications, such as mining API usages [204], improving traceability methods [33; 167], or extracting diverse models of software systems [138]. In particular, we extracted Java source code from 188 Stack Overflow posts. Concerning accuracy performance, results show that our approach accomplishes the required task with a very low number of errors, in terms of both precision and recall. Moreover, concerning time performance, our approach is able to compute the extraction in linear time; this translated, in practice, to a computation time of few seconds to parse 188 documents.[11] This means that our approach is not only effective, but also *actionable*.

11 We achieved this using the Smalltalk version of PP. The Java implementation is one order of magnitude faster `http://www.lukas-renggli.ch/blog/petitparser-java-dart`

8.9 PetitIsland: Applications

Motivated by the effectiveness of our approach applied to the extraction of Java source code fragments from NL documents, we applied it to two other scenarios: (1) classification of the lines of development emails, into source code, stack traces, patch, noise, and NL, (2) creation of a higher level parser of source code to extract only necessary information. The first application is presented in Chapter 9, here we detail the second application, which was at the technical basis of our investigation described in Chapter 4. We also envision other scenarios for application.

8.9.1 Extracting source models from system artifacts

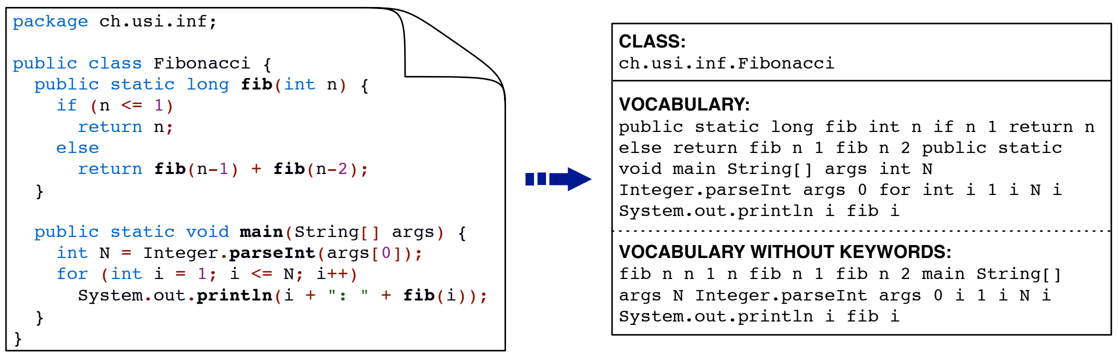
One of the first applications envisioned for the usage of island parsers is the extraction of source code models from system artifacts [136]. In fact, island parsing has the advantage, over traditional parsers, to be more robust and support the extraction of models from problematic source code. For example, island parsers can deal with source code that does not compile, is incomplete, or contains syntax and semantic errors, like undeclared variables or modules outside of the class path. Island parsers can also help with legacy source code where the grammar is not fully available; or they are useful to avoid implementing complete parsers and dealing with the intricacies of writing rules for every low level productions, when these are not necessary for the models that researchers and data scientists need to extract.

This example is another real-world application in which we successfully applied our approach to island parsing. Since PetitIsland works with arbitrary text, we can also use it for extracting customized models from source code. In Chapter 4 we dealt with the problem of recovering traceability links between emails and source code. In particular, we wanted to verify whether lightweight lexical approaches based on text matching we devised could be as effective as full-fledged IR techniques (*i.e.*, vector space model, with *tf-idf* and latent semantic analysis) for retrieving traceability links.

For this task, we decided to compare the effectiveness of the linking techniques when dealing with diverse syntaxes and naming conventions. For this reason, we considered three mailing lists pertaining to Java systems, one to a PHP system, one to an ActionScript system, and one to a C system. To conduct our comparison, we had to extract information from the source code of these systems written in four different programming languages. In particular, to apply IR techniques we had to extract a model with the name of the classes and the terms included in their definitions (as depicted in Figure 8.24), so that we could compare their vocabulary (including, or not, keywords) with that of each candidate email. The most traditional approach to extract the model is to use specialized parsers for each language and model their output. However, the specialized parsers were available in different programming languages, and generated AST in different formats that should have been visited with different procedures. Although we adopted this approach in a first attempt, we figured out the severe drawbacks of using so diverse parsing approaches for model extraction, especially in terms of maintainability and evolution.

For this reason, we devised an approach, based on PetitIsland, to extract the necessary models from the source code. This allowed us to use the same technology for the parsing of each language, thus having consistent implementation, output, and subsequent transformation procedure. This solution improved the maintainability and extensibility of our analysis.

For the purpose of conducting our analysis, we were only interested in extracting type declarations and their bodies, so that we could extract type names and the terms contained in their



```
package ch.usi.inf;

public class Fibonacci {
  public static long fib(int n) {
    if (n <= 1)
      return n;
    else
      return fib(n-1) + fib(n-2);
  }

  public static void main(String[] args) {
    int N = Integer.parseInt(args[0]);
    for (int i = 1; i <= N; i++)
      System.out.println(i + ": " + fib(i));
  }
}
```

```
ch.usi.inf.Fibonacci
```

```
public static long fib int n if n 1 return n
else return fib n 1 fib n 2 public static
void main String[] args int N
Integer.parseInt args 0 for int i 1 i N i
System.out.println i fib i
```

```
fib n n 1 n fib n 1 fib n 2 main String[]
args N Integer.parseInt args 0 i 1 i N i
System.out.println i fib i
```

Figure 8.24: Class modeling for text analysis.

body declarations and compare to the terms found in emails. For this task, it is useless to have a parser that recognizes very detailed productions, such as statements or expressions; a parser that recognizes type declarations and collect the text within their body (without parsing it) is sufficient. Implementing such a parser, with our approach to island parsing, is orders of magnitude less time consuming than implementing a full-fledged parser. We implemented four specialized parsers, one for each considered language.

For the Java programming language, we took advantage of `PPJavaIsland` and we reduced it to a minimal version. We removed all the productions more detailed than type declarations. For the other three programming languages, we wrote the parsers with a top-down approach, starting from the most comprehensive production (*i.e.*, compilation units) down to type declarations. Being type declarations very high-level productions, we managed to complete our parsers writing less than ten productions each.

To verify the quality of our parsing, we compared its output to the one generated by the specialized parsers (*e.g.*, inFusion, see Section 3.2.1). For all the cases, except Java, the result of the different techniques were matching: We have been able to replicate the output of the other parsers, by applying our approach to island parsing and using a limited number of productions. In the case of Java, our parser was able to retrieve approximately 10% more classes and definitions than the specialized parser. We informed the authors of the specialized parser about this issue and they found a bug in their modeling procedure fixed in the subsequent release.

With respect to the validation and the previous application, in this case we also show that with our approach we can not only extract the fragments in which we are interested, but also conduct *fact extraction* by modeling the fragments to exploit the information they contain, as we did with iLander (see Section 8.5).

8.9.2 Other applications

We reported on two concrete scenarios for which our approach to island parsing has proved to be effective and efficient. We believe that this approach could also be helpful in various other application contexts in the broad domain of software engineering. We briefly anticipate two:

Analysis of software under development. Since our approach can parse incomplete or incorrect source code fragments, it also opens new perspectives for software developers. The latter could benefit, for instance, from a tool that automatically relates the code just being written to natural language artifacts, such as requirements documents, documentation, emails, bug reports, chats, meeting minutes, *etc.* In this direction, we already started to use island parsing to automatically suggest related Stack Overflow questions and answers from within the Eclipse IDE, while the developer is programming [157].

Analysis of systems of systems. Our parsing approach may also prove convenient in the context of software ecosystems analysis, where several software projects, development teams and repositories are involved, and where the implicit relationships between them must be revealed in support to program understanding, bug fixing, impact analysis, or change propagation, just to name a few.

8.10 Related Work

Our approach hinges on island grammars and is inspired by a number of works that concern the extraction of source code from software engineering artifacts that contain natural language.

One of the pioneering approaches in this area is the work of Murphy and Notkin [138], who proposed a lexical approach based on regular expressions to extract models of a software system from diverse software artifacts. Software engineers can obtain consistent models from any kind of textual artifacts concerning software, by: (1) defining patterns (by using regular expressions) that describe source code constructs of interest in a software artifact, *e.g.*, function calls or definitions; (2) establishing the operations to be executed whenever a pattern is matched in an artifact being scanned; and (3) implementing post-processing operations for combining information extracted from individual files into a global model. Although the approach is lightweight, flexible and tolerant, it is based on regular expressions (to be written by practitioners) that are theoretically less powerful than a full-fledged parsing approach.

These steps are non-trivial, especially when dealing with unstructured artifacts written in natural language, such as emails. For example, choosing the best approach to expose code fragments of interest requires an accurate analysis of the advantages and drawbacks that the different regular expressions offer; moreover, what is extracted by regular expressions must be parsed to extract the meaning of the fragment. Practitioners could find it difficult and time-consuming to perform these tasks through an iterative trial and error process.

In our previous work (see Chapter 7), we created an automatic method to address the *first* step of the approach by Murphy and Notkin: We devised and evaluated lightweight techniques able to *identify* lines of Java code in emails. Since we found that the last character is a good indicator of the nature of a line, we implemented simple *lexical* rules, mainly based on pattern matching and regular expressions. For example, we were able to detect most Java lines by selecting lines ending with curly brackets or semicolons. However, this is not a parsing approach, it often loses the context between lines and it cannot be used for further fact extraction and modeling.

Island-based parsing. The use of island parsing for the automated analysis of textual artifacts is not new. Back in 1988, Stock *et. al* [182] proposed a technique for analyzing complex sentences written in natural language. The processing starts from *easily identifiable fragments* (the islands) and then extends the search scope bidirectionally to detect missing fragments in the sentence. Moonen [136] showed how island grammars may allow the derivation of robust parsers for programs written in a particular programming language. Our work is at the intersection of those two approaches; in fact, we use island grammars to offer a new approach to extracting structured elements from natural language text. Rigby *et al.* [167] performs source code extraction from natural language documents and uses it for recovering traceability links, using regular expressions to perform extraction. Our approach, using a full-fledged island parser that supports more expressive context-free structures, outperforms it in terms of effectiveness.

Source code identification/extraction from natural language artifacts. Bettenburg *et al.* devised INFOZILLA, a tool to recognize patches, stack traces, source code snippets, and enumerations in the textual descriptions that accompany issue reports [31]. INFOZILLA is composed of four independent filters, one per category, that are used in cascade to process the text. The source code snippet filter exploits an approach inspired by island parsing [136], while the other filters are based on text matching implemented through regular expressions. The authors reported results on the effectiveness of INFOZILLA in *differentiating* documents, *i.e.*, deciding whether they contain or not each category. They reached almost perfect results, and subsequently they used INFOZILLA in a work that investigated the features that are important when submitting bug reports [211]. Given the target of the INFOZILLA article (*i.e.*, classification), the authors did not conduct further code extraction and provided little details about the island parser implementation. Moreover, similarly to Rigby *et al.* [167], Bettenburg *et al.* also used regular expressions to perform extraction, thus supporting less expressive context-free structures and limiting the parser effectiveness.

Bird *et al.* [35] proposed an approach to measure the acceptance rate of patches submitted via email in open-source software projects. They classified emails with source code patches, but provided little information about their extraction techniques and no details on the evaluation benchmark. Tang *et al.* [185] addressed the issue of cleaning email data for subsequent text mining. They propose a cascaded approach to clean emails in four passes: *1*) non-text filtering, *2*) paragraph, *3*) sentence, and *4*) word normalization. Dekhtyar *et al.* [62] discussed challenges and opportunities in using text mining techniques to natural language software artifacts.

Fact extraction from source code. Basten and Klint [22] describe DEFACTO, a generic fact extraction technique based on the ASF[12]+SDF technology, which includes SGLR parsing. The technique consists in annotating the grammar of a language of interest with fact annotations. Based on those annotations, local facts are automatically extracted from *actual* source code by a generic fact extractor. Specific software analysis tasks may then start by further enriching the extracted elementary facts. In Basten and Klint's approach, each extracted *source code fact* corresponds to *one* particular syntax production. Techniques for fact extraction can benefit from our structured fragment extraction approach, as they could use different sources of analysis like natural language documents, which gives hints on a system's design decisions during development.

8.11 Summary

Software is, above all, a product by humans for humans. By having at disposal all the structured information stored in unstructured NL artifacts, such as emails, IRC chats, documentations, bug comments, we can perform more accurate analyses, and thus enrich our perception of software systems and their evolution.

We have presented two approaches to perform island parsing and mine structured information within NL artifacts. The first approach, ILANDER, is based on the ASF-SDF technology and, in

12 Algebraic Specification Formalism

three phases, *extracts* structured fragments embedded in NL text, *parses* their content, and *models* the information according to a meta-model that describes the structure of object-oriented systems. We evaluated each phase by applying our approach to a statistically significant set of emails, and reached very high accuracy values, despite the noisy nature of email content. We shows that the system model extracted from NL documents can be used to conduct novel system analyses that better approach the developers' point of view. For example, we can compare the impact of a change as envisioned by developers discussing it in emails to the actual implementation. We applied iLander to thestream of development emails in the ArgoUML mailing list, and extracted a comprehensive system model, which we used to analyze aspects of ArgoUML.

iLander does not come without its limitations, especially with respect to performance, flexibility, and extensibility. For this reason, we devised a second approach, which is implemented as an island parsing framework: PetitIsland. To make PetitIsland flexible, accurate, and extensible, we exploited different parsing methodologies: PEGs, scannerless parsers, and parser combinators. We explained how each of these methodologies is embedded in our framework and their usefulness. We implemented PetitIsland in a working tool (within our Miler toolset), written in Smalltalk, by using the parser framework PetitParser.

The architecture of our framework PetitIsland is based on pluggable extensions: Users can plug different parsers for the specific structured data fragments they have to analyze. To validate our approach, we implemented a parser for Java source code fragments, we plugged it in our framework, and we used this composition to extract source code fragments from 188 Stack Overflow posts. We reached very high accuracy values, despite the intricacies of Java grammar specification. Moreover, the approach has computational costs that make it scalable.

We showed how we used PetitIsland to extract ad-hoc models from source code artifacts by implementing tiny parsers for the target languages. Another application is presented in Chapter 9. These applications show how the approach can be used in practice and demonstrate its extensibility. Additionally, we envisioned other possible applications of our approach. The complete implementation of PetitIsland, together with clarifying examples and the experiment dataset, are available on the website `http://miler.inf.usi.ch/PetitIsland`.

Reflection. We claimed that recognizing the structured parts embedded in textual development artifacts, such as emails, is essential for conducting an appropriate mining and extract information that is useful for program understanding and software development. Nevertheless, we also mentioned that development emails are also made of a large amount of irrelevant information, or noise. The techniques presented in this chapter are not able to deal with this additional noise, because only focus on certain fragments. In the next chapter, we make a step further and we employ our PetitIsland approach in conjunction with machine learning to fully classify the content of development emails, so that also noise can be removed and natural language sentences can be correctly recognized.

Chapter 9

Classification of Lines in Development Emails

In the previous chapter we presented techniques for recovering structured fragments embedded in natural language text. In this chapter, we show how they can be used together with machine learning to achieve a precise classification of the lines of development emails. In this way, it is possible to know exactly which "languages" compose a text and apply to most appropriate techniques for extracting reliable information from them.

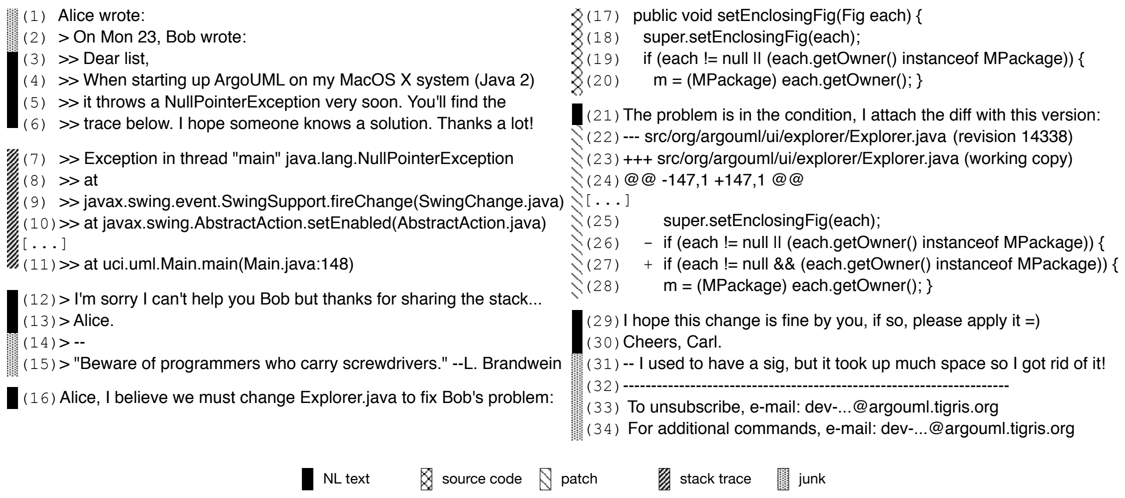
9.1 Overview

We present a work for advancing the analysis of the contents of development emails. We argue that we should not create a single bag with terms indiscriminately coming from natural language parts, code fragments, email signatures, patches, *etc.* and treat them equally. We need to recognize every language in an email to enable techniques exploiting the peculiarities of each category. This work contributes to a deeper and more detailed analysis of email communication among developers and is also applicable to similar "unstructured" data.

We propose an approach, based on a combination of parsing techniques and machine learning methods, to classify the contents of development emails in five categories: natural language, source code, patch, stack trace, and junk (text that does not add valuable information, such as auto-generated disclaimers, authors' signatures, or erroneous characters). Our technique works at the line level, which—by inspecting hundreds of emails—we found to be the appropriate granularity for email content classification. We created a web application to manually classify email content in the chosen categories. We classified a statistically significant set of emails from four Java OSS systems, used to evaluate the accuracy of our approach.

Contributions of the chapter. In this chapter, we present the following contributions:

- *We identify that development emails are composed of a number of languages that have to be recognized to enable subsequent ad-hoc analyses.* We motivate the importance of the work presented in this chapter in the next section.
- *We create a novel approach that fuses island parsing and machine learning techniques for classification of email lines.* Our approach, named Mucca, is able to perform automatic classification of the content of development emails into five language categories: natural language text, source code fragments, stack traces, code patches, and noise.
- *We create a web application to manually classify email content.* We extend the Miler Game by creating a novel user interface and adding new features.



```
(1)  Alice wrote:
(2)  > On Mon 23, Bob wrote:
(3)  >> Dear list,
(4)  >> When starting up ArgoUML on my MacOS X system (Java 2)
(5)  >> it throws a NullPointerException very soon. You'll find the
(6)  >> trace below. I hope someone knows a solution. Thanks a lot!

(7)  >> Exception in thread "main" java.lang.NullPointerException
(8)  >> at
(9)  >> javax.swing.event.SwingSupport.fireChange(SwingChange.java)
(10) >> at javax.swing.AbstractAction.setEnabled(AbstractAction.java)
[...]
(11) >> at uci.uml.Main.main(Main.java:148)

(12) > I'm sorry I can't help you Bob but thanks for sharing the stack...
(13) > Alice.
(14) > --
(15) > "Beware of programmers who carry screwdrivers." --L. Brandwein

(16) Alice, I believe we must change Explorer.java to fix Bob's problem:

(17)  public void setEnclosingFig(Fig each) {
(18)    super.setEnclosingFig(each);
(19)    if (each != null || (each.getOwner() instanceof MPackage)) {
(20)      m = (MPackage) each.getOwner(); }

(21) The problem is in the condition, I attach the diff with this version:
(22) --- src/org/argouml/ui/explorer/Explorer.java (revision 14338)
(23) +++ src/org/argouml/ui/explorer/Explorer.java (working copy)
(24) @@ -147,1 +147,1 @@
[...]
(25)       super.setEnclosingFig(each);
(26)    -  if (each != null || (each.getOwner() instanceof MPackage)) {
(27)    +  if (each != null && (each.getOwner() instanceof MPackage)) {
(28)         m = (MPackage) each.getOwner(); }

(29) I hope this change is fine by you, if so, please apply it =)
(30) Cheers, Carl.
(31) -- I used to have a sig, but it took up much space so I got rid of it!
(32) --------------------------------------------------------------------
(33)  To unsubscribe, e-mail: dev-...@argouml.tigris.org
(34)  For additional commands, e-mail: dev-...@argouml.tigris.org
```

Figure 9.1: Example development email with mixed content

- *We produce a benchmark to evaluate the classification of email lines into five categories,* i.e., *natural language, source code, patch, stack trace, and noise.* Our benchmark features more than 1,400 emails comprising almost 69,000 manually classified lines.

Structure of the chapter. In Section 9.2 we motivate our work and show how it can improve software analyses. In Section 9.3 we show how we collected and manually annotated the email data. In Section 9.4 we detail our classification methods and their evaluation. We discuss threats to validity in Section 9.5. In Section 9.6 we describe the related work. We summarize our contributions and conclude in Section 9.7.

9.2 Motivation

Figure 9.1 shows the body of an example development email. Due to the variety of languages used, if we consider the content of such email as a single bag of words, we will obtain a motley set of flattened terms without a clear context, thus severely reducing quality and amount of available information. Inversely, by automatically distinguishing the parts composing the email, we support many tasks, such as:

Traceability Recovery. In Figure 9.1, the email is referring to several classes (*e.g.*, `Main`, `Fig`, and `MPackage`), but only the class `Explorer` is critical to the discussion: It causes a failure and the email's author changed it to provide a solution. We realize the importance of `Explorer` by reading the NL line 16. While analyzing emails for program comprehension (see Chapter 6), we often found this pattern: Artifacts mentioned in natural language parts of emails are more relevant to the discussion than artifacts mentioned in other contexts (*e.g.*, stack traces). A traceability method based on bags of words (*e.g.*, those presented in Chapter 4) cannot recognize whether references to artifacts appear in a natural language context, to increase the link relevance. Such a method can only use the number of occurrences to weight more certain terms [123], leading to imprecise results. In Figure 9.1, a weighting based on occurrences would give the most relevance to class `MPackage` (mentioned 5 times), which is actually marginal to the discussion. *By recognizing the context in which a term appears, one can elicit weights for words appearing in a document dynamically and more accurately, improving the traceability links' quality and giving more information to the user.*

Stop Words Removal. To better characterize documents, IR research invites to remove *stop words*, *i.e.*, very common words [123], thus weighting more the peculiar terms of a document. This approach is less beneficial when applied to development emails: By removing stop words, one reduces the noise in natural language parts, but also deletes information in parts with a different vocabulary (*e.g.*, source code). For example, deleting the stop word "each" from the content of Figure 9.1 means also deleting a variable name in a code fragment (lines 17–20) and a patch (25–28). This is suboptimal, since variable names provide relevant information [111]. Similarly, we delete important information when we remove programming language keywords from natural language. *By recognizing the different parts that compose an email, one can use different common terms removal techniques, thus exposing the most relevant information.*

Artifact Summarization. Due to the amount of data produced during a system's evolution, researchers investigated how to expose only the significant parts to reduce information overload (*e.g.*, [160]). The proposed techniques are tailored to specific types of artifact (*e.g.*, code [89], natural language documents [99]) and cannot be applied to mixed documents, such as emails. *By recognizing the different parts of an email, one can use the most suited summarization technique according to each part's type and extract correct information.*

Fact Extraction. To know the facts expressed in code fragments, patches, or stack traces, one can use ad hoc parsers. In Figure 9.1, using a parser for patches, one recognizes that the file being modified is `Explorer` (lines 22–23). Similarly, natural language text can be analyzed with NLP techniques [101]. However, ad hoc parsers cannot be applied to mixed content, because they are not robust enough to manage unexpected data. *By distinguishing the type of each email line, we can exploit ad hoc analysis techniques to extract precise information.*

Non-essential Information Removal. In Figure 9.1, 8 lines out of 34 contain irrelevant data–"junk". Previous research indicated that some changes in version history are not essential, and that their detection and filtering can improve change-based analysis techniques [102]. Similarly, the detection and removal of junk from email content increase the data quality [32], thus improving the quality of analyses. *By recognizing the noise in emails, the important data emerges, improving the information extraction quality.*

9.3 Data Collection and Classification

Since we strive for devising a method for reliably and precisely classifying email lines, with the aim of improving data quality and comprehension, we need data sets that are *accurate*, *comprehensive*, and of *statistically significant* sizes. This is critical for the validation and leads to more reliable training for the supervised classification methods. To this aim, we extended the Miler Game (see Section 3.2.1) to assist the manual classification of email content in categories.

9.3.1 Data Collection

Different software systems often use different applications to manage email repositories. We tackled this using our issue by importing data using our Miler importer for MarkMail (see Section 3.2.1).

Table 9.1: Email data sets used in the experiment, by system

System	Emails		
	Population	Filtered	Sample
ArgoUML	25,538	25,538	379
Freenet	23,134	23,134	378
JMeter	24,005	5,814	361
Mina	21,384	14,499	375
Total	**94,061**	**68,985**	**1,493**

Table 9.1 shows the four software systems and mailing lists we considered (for more information on the systems see Section 3.4). We selected unrelated systems emerging from the context of different free software communities, *i.e.*, Apache, ArgoUML, and Freenet. The development environment and paradigms, and the usage of the mailing lists are likely to differ, thus mitigating external validity threats. We imported all the messages starting from the mailing list inception (second column in Table 9.1) to the end of 2010. The only pre-processing conducted on the emails was filtering out messages automatically generated by bug tracking systems and versioning systems, since they have an easily parseable structure that can bias the results.

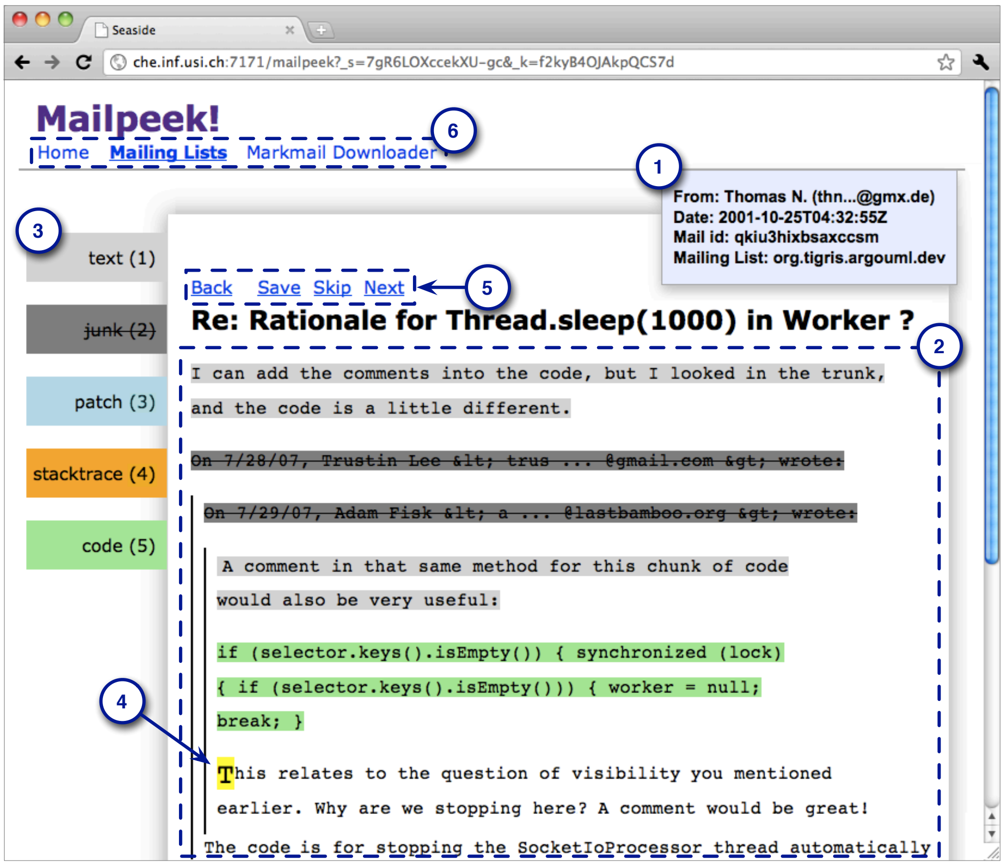
From each filtered mailing list, we extracted statistically significant sample sets (last column, Table 9.1), which were used by the approach without any pre-processing on the text. Since we had no prior knowledge on the distribution of line categories in the populations, we opted for simple random sampling [193] to pick the emails. The chosen sizes have a 95% confidence level and a 5% error margin (see Section 4.2 for more information about sample size determination).

9.3.2 Data Classification

To test our approach and train supervised machine learning classifiers, we needed to manually classify the 1,493 sample emails. To ease this manual task and alleviate its error-proneness, we devised MailPeek, a web application written in Smalltalk and embedded in our Miler toolset (see Section 3.2.1).

Figure 9.2 shows MailPeek's main window, as it appears in a web browser after a user selects a mailing list of interest and the application extracts a random email among those not automatically filtered. MailPeek displays the email metadata (Point 1), and content (Point 2) with vertical bars to show indentation levels and increase readability.

Users conduct the classification task at the *character* level: To label a block, they (1) click on starting and ending characters, (2) verify the correctness of the selection (which is shown in a yellow background), and (3) apply the appropriate category, either by clicking on a button in the left menu (Point 3), or by using keyboard shortcuts. The character granularity provided us



From: Thomas N. (thn...@gmx.de)
Date: 2001-10-25T04:32:55Z
Mail id: qkiu3hixbsaxccsm
Mailing List: org.tigris.argouml.dev

Re: Rationale for Thread.sleep(1000) in Worker ?

```
I can add the comments into the code, but I looked in the trunk,
and the code is a little different.

~~On 7/28/07, Trustin Lee &lt; trus ... @gmail.com &gt; wrote:~~

 ~~On 7/29/07, Adam Fisk &lt; a ... @lastbamboo.org &gt; wrote:~~

  A comment in that same method for this chunk of code
  would also be very useful:

  if (selector.keys().isEmpty()) { synchronized (lock)
  { if (selector.keys().isEmpty())) { worker = null;
  break; }

  This relates to the question of visibility you mentioned
  earlier. Why are we stopping here? A comment would be great!

 The code is for stopping the SocketIoProcessor thread automatically
```

Figure 9.2: Mailpeek: our web app for classifying email content

the basis to decide which granularity was appropriate for the automatic classification, *i.e.*, line granularity (see Section 9.3.3).

When users hover with the mouse on any character in the email content area (Point 2), the character font size triples (Point 4). According to Fitts' Law [119], this eases the selection, thus decreasing fatigue and errors.

Once an email is completely classified, the user clicks on `save` (Point 5) and MAILPEEK loads another random email among those not yet classified. The `skip` link allows the user to leave out non-valid emails that were not removed by the automatic filtering phase. The top menu (Point 6) allows users to change mailing list or trigger the importer.

Two graduate students from the REVEAL Research Group at the University of Lugano, with several years of JAVA programming experience, conducted the manual classification task on two distinct sets of emails. We evaluated the inter-rater agreement by asking them to also classify 5% of the emails analyzed by the other person. In this sample, we found 12 non-concordant lines (less than 0.2%).

9.3.3 Data Distribution

Table 9.2: Distribution of the categories per line, by system

		ArgoUML		Freenet		JMeter		Mina		Total	
NL Text	■	10,945	47.2%	7,923	59.6%	7,778	41.8%	6,496	51.2%	33,142	48.9%
Junk	▦	11,122	47.9%	4,096	30.8%	9,734	52.3%	4,633	36.5%	29,585	43.6%
Patch	⧅	470	2.0%	986	7.4%	339	1.8%	287	2.3%	2,082	3.1%
Source Code	⊠	304	1.3%	29	0.2%	591	3.2%	990	7.8%	1,914	2.8%
Stack Trace	▨	364	1.6%	254	1.9%	165	0.9%	286	2.3%	1,069	1.6%
Total		23,205		13,288		18,607		12,692		67,792	

Table 9.2 reports categories' distributions in the sample sets. Most lines are NL; more than 30% of lines are junk, thus stressing the impact of noise on email data; the frequency of other categories is lower and the ranking changes according to the mailing list. The different composition of the email sets' contents reflects the different usage of mailing lists among the communities. Some lines are *hybrid*: they belong to more than one category, and are mostly composed of junk not separated by the NL text. They account for less than 5% of the population (*i.e.*, 3,362 lines). To mitigate the bias in the experiment we include them as separated instances.

9.4 Experiment

We created a number of techniques based on ideas gathered both from the IR field, which we reshaped and adapted, and from language programming parsing. Even though the techniques can be used in isolation, we achieved the best results by creating a unified approach (detailed in Section 9.4.5).

9.4.1 Term Based Classification

"*Most current IR systems are based on a kind of extreme version of compositional semantics in which the meaning of a document resides solely in the set of words it contains*" [101]: In IR systems, documents are considered as bags of words, where syntactic information, ordering, and constituency of the words play no role in determining their meaning. In practice, this is the same as vector space modeling, which we used in Chapter 4: Each document is modeled as a vector of features, which correspond to *terms* in the corpus vocabulary. For example, if we consider a document (d), the cardinality of the vocabulary ($|C|$), and how many times each term (t_i) occurs in d, we could define the document vector as in Equation 4.5.

This vector modeling has been widely used with supervised machine learning algorithms to achieve very effective results in automatic text classification [123; 176]. We ground the first techniques on the same basis: We consider lines as vectors of terms and use machine learning for their classification (as in Figure 9.3)

$$\begin{array}{lcccccc} & \mathbf{t_1} & \mathbf{t_2} & \mathbf{t_3} & \mathbf{t_4} & \cdots & \mathbf{t_C} \\ \mathbf{L_1} & 0 & 1 & 3 & 0 & \cdots & 0 \\ \mathbf{L_2} & 0 & 0 & 1 & 0 & \cdots & 2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ \mathbf{L_N} & 0 & 0 & 1 & 2 & \cdots & 1 \end{array}$$

Figure 9.3: Lines modeled as vector of term-features

In the following we describe and motivate our choices in terms of the used machine learning technique and vector features (*i.e.*, terms), which cannot be based on results from the IR field, as they refer to other domains and classification tasks.

Machine-Learning Method. We employ Naïve Bayes, a method of *supervised* learning (*i.e.*, machine learning algorithms that use classified training examples to infer the classification function). Naïve Bayes relies on the *conditional independence* assumption: The presence of a feature is unrelated to the occurrence of the other features. Even though the assumption is a strong simplification, the method often outperforms more sophisticated techniques [101]; in particular in text classification, Naïve Bayes achieves significant results [42]. An asset of Naïve Bayes is its linear complexity, which allows training and classification to be performed efficiently, even with a very large number of features.

The method uses Bayes' rule [101] to compute the probability that a line l, made of t_k terms, belongs to class c:

$$P(c|l) \approx P(c) \prod_k P(t_k|c) \tag{9.1}$$

It computes the posterior probability $P(c_i|l)$ for each class and chooses the one with the highest probability. This is the *maximum a posteriori* (MAP) hypothesis [101]:

$$C_{MAP} = \arg\max_{c_j \in C} P(c|l) \approx \arg\max_{c_j \in C} P(c) \prod_k P(t_k|c) \tag{9.2}$$

If we want to classify the line $d = ''Alice\ wrote\ :''$ as $text$, $junk$, or $code$, the algorithm first computes the probabilities as: $P(text|l) = 0.43$, $P(junk|l) = 0.55$ and $P(code|l) = 0.02$, then selects the value 0.55, thus classifying l as $junk$.

Given the high number of probability multiplication performed, the calculated values may become too small to be represented by float numbers: This may introduce the risk of underflow. To avoid this issue, Naïve Bayes computes the values as logarithms. Moreover, when a term does not occur in the training test, the calculated probability would be zero, thus Naïve Bayes also applies a Laplacian smoothing to the product.

Selection of the Terms. The selection of the terms is essential to the right functioning of this approach. This is how we considered the terms:

Words. They are the fundamental tokens of all the languages we want to classify. We judge the words in our corpus of 67,792 non-empty lines to be proper features for line modeling. Contrarily to most IR methods, we do perform neither *stop word removal* (*i.e.*, excluding very common words), as we expect very frequent words to be representative of a class (*e.g.*, Java keywords in code), nor *stemming* (*i.e.*, collapsing the morphological variants of a word), as we expect some variants to be more characteristic of certain classes (*e.g.*, verb tenses in NL text).

Punctuation. We must distinguish lines written in languages with different syntaxes, thus we consider punctuation to be a valuable aspect. Unless the punctuation marks are separated by words or spaces (*e.g.*, the dots in `javax.swing.`, are two occurrences of the feature `javax.swing.`), we consider them as a single term, thus recognizing special characters, such as `javax.swing@@` in line 24 in Figure 9.1. We do not consider email reply threading characters (*e.g.*, `>` and `>>` in lines 2-15 in Figure 9.1) at this point, as they do not have a definite role for line classification.

Bi-grams. Naïve Bayes relies on the conditional independence assumption, which makes the modeling of NL text features feasible. However, the other considered languages have a stricter syntax where patterns of terms appear together (*e.g.*, "*public void*" in code). To model this dependency characteristic of some terms, thus also reducing the negative effects of Naïve Bayes' assumption, we also consider *bi-grams* (*i.e.*, pairs of terms appearing one after the other).

Context. All the features considered so far are extracted only from the line under classification. However, some of the considered classes (*i.e.*, patch and stack trace) have a structure recognizable only by considering surrounding lines. For example, line 18 and line 25 have the same content, thus can be mapped to the correct class only considering the context lines. Researchers proposed to solve a similar problem by adding features with characteristics of lines close to the one under classification [43; 185]. We adapt this approach to our case and consider what appears in the preceding and following lines. For example, in addition to `@@`, we have the features `@@-lineBefore`, and `@@-lineAfter`.

Table 9.3: Results with term based classification, by feature sets

	Number of Features	10-fold cross validation			Mailing list cross validation		
		Correct Lines		Impr. sig.	Correct Lines		Impr. sig.
Words	12,658	46,555	68.6%		46,056	67.9%	
Words, Punctuation	19,384	62,938	92.8%	$p < 0.001$	58,172	85.8%	$p < 0.001$
Words, Punctuation, Bi-grams	145,187	63,413	93.5%	$p < 0.001$	58,568	86.4%	$p < 0.001$
Words, Punctuation, Bi-grams, Context	435,561	63,708	93.9%	$p < 0.001$	60,580	89.4%	$p < 0.001$

Line Modeling. After defining the aforementioned features, we modeled each line as vector a of $n + 1$ dimensions. The first n elements are the chosen features, while the last one is the manual classification value (*e.g.*, `patch`). The first column of Table 9.3 shows the values of n according to the considered subset of features. Each feature is populated with the corresponding term's occurrences in the line (*e.g.*, if the feature t_i stands for the term "*public*", and the line l contains two occurrences of it, then in v_l, we have $t_{i(l)} = 2$). When a line contains terms that are not mapped as feature, they are discarded.

9.4.2 Training and Testing

Since we use a supervised machine learning algorithm, we need to train it on classified data. We use two different approaches for training the model and show how this affects the results when testing of the model's accuracy. To evaluate the model's accuracy, we count the number of correctly classified lines and we use the IR metrics presented in Section 4.2.4: *precision* (Equation 4.2), *recall* (Equation 4.3), and *F-measure* (Equation 4.4). TP (true positives) are correctly classified lines, FP (false positives) are not correctly classified lines.

Ten-Fold Stratified Cross-Validation. As a first step, we apply 10-fold stratified cross validation [193]: We split the dataset in 10 folds, use 9 folds (90% of the lines) to train the prediction model, and use the remaining fold to test the model's accuracy. This process is repeated 10 times rotating the training and testing folds. The distribution of classes is kept equal in training and test sets. Columns 2 and 3 in Table 9.3 show the results. Each subset of features adds information that increases the results in a significant way (column 3). When considering all the features, the ratio of correctly classified instances reaches almost 94%.

Mailing List Cross-Validation. Different mailing lists discuss about different systems and are likely to use different words and jargon. For example the mailing list signature (*e.g.*, lines 32–34 in Figure 9.1) have different terms in each mailing list. Thus, term-features that work for one mailing list may not be useful for others. To better test the generalizability of the results achieved by the classifier, we conduct a "mailing list cross validation." In practice, it is a 4-folds cross validation, in which folds are neither stratified nor randomly taken, but correspond exactly to the different mailing list: We train the classifiers on three mailing lists and we try to predict the classification of the remaining mailing list. We do this four times rotating the mailing lists and we measure the average results. Columns 4 and 5 in Table 9.3 show the results.

As expected, testing with mailing list cross validation, the classifier performance drops, even when considering all the features. However, this is a more relevant test to understand the results of the classifier applied to unseen Java development mailing lists, and we use it in following.

Table 9.4 reports the confusion matrix [123], precision, recall, and F-measure values for the classification with all the term-features (*i.e.*, words, punctuation, bi-grams, and context). The best results are achieved in classifying text, junk, and stack trace, while patch and code are often misclassified among themselves. This is reasonable, since recognizing those lines requires a large context: Even a human reader cannot determine to which class line 28 in Figure 9.1 belongs without inspecting many lines. However, differentiating code and patches might be useful for various tasks, such as improving traceability links or automatically estimating the topic and purpose of the email (see Section 9.2).

Table 9.4: Mailing list cross validation on the best set of features

classified as →		NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	■	32,062	1,046	20	8	6	0.894	0.967	0.929
Junk	▒	3,269	26,225	54	14	23	0.942	0.886	0.913
Patch	▧	207	343	946	585	1	0.452	0.454	0.453
Source Code	▩	309	121	1,074	410	0	0.403	0.214	0.280
Stack Trace	▨	35	97	0	0	937	0.969	0.877	0.920

9.4.3 Term Based Features and Overfitting

By considering the entire set of features (*i.e.*, words, punctuation, bi-grams, and context), we obtain a complex classification model with more features than training instances. In such a scenario, *overfitting* is likely to occur—this hypothesis is supported by the reduced performances of the classifier in mailing list cross validation (see Table 9.3). By reducing the features that are not valuable to correctly predict instances outside the training set, we decrease overfitting and increase the generalizability of the results.

Since we use words and punctuation to describe the common traits of each language, we hypothesize that the terms that rarely occur in the corpus are less relevant and can be removed. We investigate this hypothesis by gradually filtering out features (from all four kinds) that appear in less than t lines and inspecting the results. We consider all the values of t from 1 to 22,769 (*i.e.*, lines in which the most occurring term `.` appears).

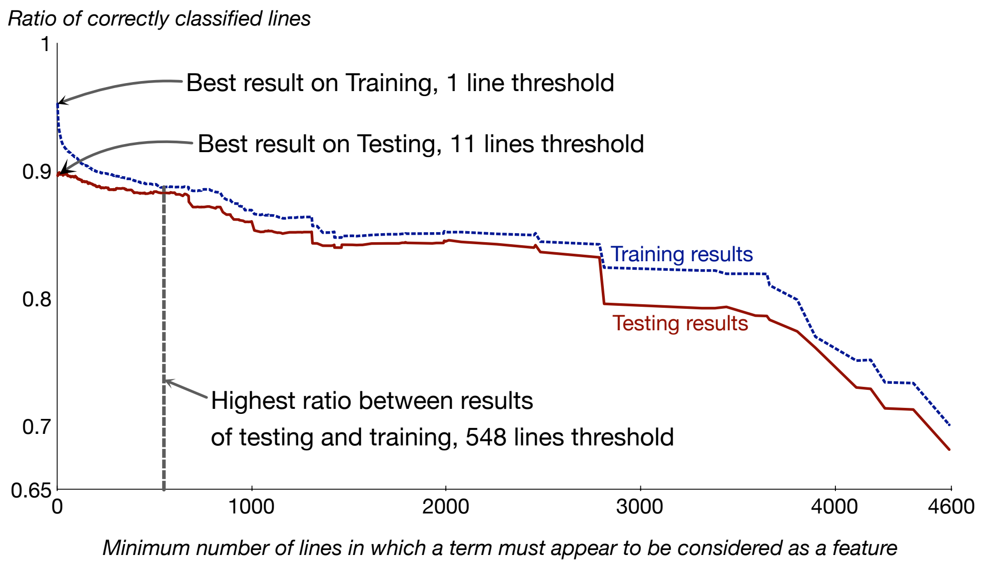


Figure 9.4: Results on training and test sets, by line threshold for features

Figure 9.4 shows the average classifier's performance in mailing list cross validation, with t ranging from 1 to 4,587 (higher values reduce the number of features to less than 10 greatly reducing the results). The blue dashed line (above) is the average percentage of correctly classified lines on the training set, while the red solid line (below) is the average percentage on the test set. The best result on the training set (*i.e.*, 96.1%) is set at t value of 1, (*i.e.*, we consider all the features, 115,864 on average when training on three mailing lists), while the best result on the testing set (*i.e.*, 89.9%) is set at t value of 11 (*i.e.*, 5,618 features on average), which reduces some noise. The optimal t value for the best testing set results, however, changes according to the mailing list: Two lists have a t value of 2, one of 25, and one of 46. A valid approach to find a good value for t, also for unseen data, is to consider the point with the highest ratio between testing results and training results [193]. We find this hot spot with a threshold of 548 lines (*i.e.*, 122 features on average). Interestingly the number of features is a tiny fraction of the initial ones, but the testing results are reduced only by a 1.5% (*i.e.*, 88.3%). Higher thresholds lead to lower performances.

9.4.4 Parsing Based Classification

We tackle the classification from a different perspective and use a different approach: parsing. In fact, three of the considered classes (*i.e.*, stack trace, patch, and source code), which are either produced or consumed by a machine, present a clearly structured and defined syntax that may be recognized even if embedded in a noisy unstructured context. We use PetitIsland described in Chapter 8 to write a specialized parser per each class (excluding NL text), based on the concept of island parsing [136]. We detail only the most salient features of each parser. The complete source code is available at `http://mucca.inf.usi.ch`.

Stack trace island parsing. To illustrate how we implemented island parsing of stack traces, we define some terminology to refer to the various parts of their structure. Consider, for example, Figure 9.1:

- the `exceptionMessage` refers to the natural language message usually included at the beginning of stack traces (*e.g.*, line 7);
- the `atLine` refers to a line that reports a method invocation occurred in a specific file (*e.g.*, lines 8–11);
- the `ellipsisLine` is a line used to reduce lengthy stack traces and has the form: "*…<number> more*";
- the `causedByLine` is a line that might appear at any point in a stack trace to introduce a new nested trace and has the form: "*Caused by: <stacktrace>*".

We defined a parser class for parsing stack traces:

```
PPCompositeParser subclass: #PPStackTrace
  instanceVariables: 'stackTrace stackTraceLine atLine ellipsisLine [...]'
```

Among the productions, we defined `atLine` and `ellipsisLine` because they have the most recognizable form. By plugging `PPStackTrace` into a new instance of `PPIsland` and testing our approach on the whole corpus we found no errors in extracting these parts of the stack trace:

```
PPStackTrace>>atLine
  ^at , qualifiedMethod ,
    leftParenthesis , classFile ,
     ((colon , number)
     / (comma, compiledCode)
     / (leftParenthesis , compiledCode , rightParenthesis)) optional ,
   rightParenthesis

PPStackTrace>>ellipsisLine
  ^ellipsis, number, more
```

The `exceptionMessage` and the `causedByLine` elements have a mostly unpredictable structure (*e.g.,* different JAVA virtual machine versions may output the same error message differently), thus they cannot be parsed with a specific grammar. To overcome this issue we use a double-pass approach: In the first pass, we recognize and mark all the occurrences of `atLine` and `ellipsisLine`; in the second pass, we look for each line that contains strings such as "exception", "error", "failure", *etc.* When such a line exists, if the next n lines belong to those lines marked in the first step, we classify it and all the lines up to the first `atLine` as `stack trace`. We empirically found the n value equals to 3, to be a good tradeoff between precision and recall.

For example, if we apply our stack trace parser to the email in Figure 9.1, in the first pass, it would classify lines 8–11 as *stack trace*; in the second pass, it would consider lines 5 and 7 as `exceptionMessage` candidates, since they both contain the string "*exception*". Finally, it will only pick line 7, because in the next 3 lines there is an *atLine* element (in this heuristic, we also count the empty lines, such as the line between 6–7).

Since this method does not require training, we tested it on the all manually classified instances, reaching an F-measure value of 99.1% in the classification of stack trace lines. The complete results are reported in the first row of Table 9.5.

Patch Island Parsing. For the patch parser, we also define some terminology for their typical structure. Again, consider Figure 9.1:

- the `patchHeader` refers to the first two lines of a patch, which contain the reference to the modified file and, optionally, the revision versions (*e.g.,* lines 22–23);
- the `patchBlockHeader` refers to the lines detailing the modification done by the patch on a chunk (*e.g.,* line 22);
- the `patchBlock` refers to all the lines in the chunk (*e.g.,* lines 25–28).

A single patch has only one `patchHeader`, while it might have multiple occurrences of `patchBlockHeader` followed by the respective `patchBlock`.

We devised a parser, `PPPatch`, adopting an approach similar to the one of the stack trace parser: We started from the most recognizable lines and expanded to include the more ambiguous ones. The parsing is done in a single pass: We wrote a production for the `patchHeader`, even if split on multiple lines, by using the tokens `---`, `+++`, and `@@` as hooks; then we generated a parser that first recognizes the `patchBlockHeader` (thanks to its clear structure), then matches the following `patchBlock`. The patch blocks are problematic, since they have variable length and their ending is not clearly defined. In fact, after the deleted and added lines (which are marked with initial

`+` or `-` signs, as in lines 26–27), patches include *some* contextual lines: Their number may vary between zero and three, or more if not well formatted. Bird *et al.* tackled the patch block ending issue both by using the information about the range to be found in the `patchBlockHeader` and by analyzing how a line starts (usually the context lines should be preceded by a space) [35]. However, in our dataset we found this information to be not reliable, because of unexpected line breaks and wrong formatting. For this reason, we implemented a lookahead heuristic that checks whether the lines after the `+` or `-` signs might be good candidates as patch. The heuristics checks whether the lines are source code, by using a reduced version of the `PPJavaIsland` parser, and in the positive case it classifies them as *patch*.

The complete results are reported in the second row of Table 9.5. As expected, since we used a conservative lookahead threshold (maximum four lines), we have a higher precision and lower recall. By manually inspecting the false negatives, we noticed that the low recall is also due to some patch lines that have neither `patchHeader` nor `patchBlockHeader`, thus resulting ignored by our parser.

Source Code Island Parsing. Among the three classes with structured language (*i.e.*, stack trace, patch, and source code), code is the most ambiguous. We used a preliminary version of the `PPJavaIsland` presented in Section 8.8. We note that our island parser for source code would match most of the content of a `patchBlock`, because they do contain valid source code. This increases the number of false positives. For this reason, we chain the source code parsing to the patch parsing: We first detect the patches, then, on the lines that are *not* classified as patch, we run the source code parser. As a beneficial side effect, this chained procedure reduces the text and the ambiguities to be managed by the island parser, thus increasing the performances. The complete results are reported in the third row of Table 9.5.

Junk Parsing. Noisy text, such as authors' signatures, is hard to automatically distinguish from NL text; however, some peculiar common patterns can be matched with a parser. This approach is made of three steps: (1) matching and classification of email headers (*e.g.*, lines 1 and 2 in Figure 9.1) with a regular expression; (2) identification and extraction of signatures of mailing lists (*e.g.*, common lines added to the end of every email sent to the same list, such as lines 32–34) and authors; and (3) usage of the recognized signatures to automatically compose a grammar for generating a parser to match them, under any possible formatting or position in the email body. To recognize signatures, we consider all the emails whose last block of text is not quoted from previous emails (this can be easily achieved by considering lines that do not end with email reply threading characters, such as `>` and `>>` in lines 2-15 in Figure 9.1). In these emails, the authors themselves conclude the message and most probably include their signatures. For example, the email in Figure 9.1 contains the author's signature in the last block. Among the selected emails, we only consider the last not quoted block. We analyze it backward starting from the last line (*e.g.*, from line 34 up to 16). When we encounter a line that starts with, or is only composed of, two or more dashes, underscores, or stars, we take out the lines up to the bottom and consider it as a signature. The process continues up to the top of the not quoted block. For example, the algorithm applied to the email in Figure 9.1 extracts lines 32 to 34, and line 31 as signatures.

By classifying these blocks as junk, we would miss the cases in which signatures are in quoted text (*e.g.*, lines 14–15). We, thus, conduct the third step: We automatically define a grammar from each extracted string able to recognize the signature in any possible position or formatting the text; then, we use these grammars to automatically generate the relative parsers; finally we classify matched lines as *junk*. This approach reaches an F-measure value of 81.2%, by recognizing more than 65% of the junk lines.

Results. Table 9.5 reports the results of each parser in the classification of the lines into the corresponding type. For example, the first line covers the results in using the Stack trace parser to classify lines as *stack trace*. The false positives (*e.g.*, 4 in the first row) are lines classified as *stack trace* by the method, but with a different manual classification.

Table 9.5: Single classification results achieved by using parsers

	Total Instances	TP	FP	Precision	Recall	F-Measure
Stack trace parser	1,069	1,054	4	0.996	0.986	0.991
Patch parser	2,082	1,996	0	1.000	0.959	0.979
Source code parser	1,914	1,715	74	0.959	0.896	0.926
Junk parser	29,585	20,372	226	0.989	0.689	0.812

All the parsers reach high classification values, while being mailing list independent and requiring no training. However, parsers have limitations: (1) They are manually implemented, and for this reason they cannot predict or cover all the possible variants of the patterns that they match, especially due to truncated content; (2) the values reached in classifying junk are lower than those achieved with the machine learning approach.

We argue that these parsers are not only valuable thanks to the high classification values they achieve, but also because they are mailing list independent and require no training to be used. The former feature is given by the fact that the parsers rely on syntactical characteristics of programming languages, stack traces, and patches, that are the same across all the mailing list pertaining to JAVA systems; the latter feature allows their usage on any textual source of data.

Next, we present a method that overcomes these issues by fusing machine learning and parser-based approaches and create a more complete, precise, and robust approach.

9.4.5 Unified Approach

This approach fuses the term based classification and the parser-based approaches.

Adding Parsing Results to Naïve Bayes. Naïve Bayes is not limited to use *terms* as features: One can include any relevant aspect as a feature in the classification process. Given these premises, we add the parser-based classification output to improve the Naïve Bayes machine learning process. We do this by adding four new features to the feature-vectors, in addition those presented in Section 9.4.1. Each new feature maps the output of a parser: The value is 1 when the corresponding parser matches the specific line, 0 otherwise. We used Naïve Bayes and performed mailing list cross validation. For example, Tang *et al.* considered the quotation level in which an email line resides as a valuable feature for recognizing noise [185].

Given these premises, our intuition is that we can add the output of the parser-based classification to improve the Naïve Bayes machine learning process. We do this by adding four new features to the feature-vectors, in addition to words, punctuation, and bi-grams. Each new feature maps the output of a parser: Its value is one when the corresponding parser-based classifier matches the specific line, and it is zero in the other cases. For example, we expect the feature-vectors corresponding to lines 31 to 34 in Figure 9.1 to have the feature `junk parser` with value one, while the other parser-based features with value zero; similarly, we expect the "stack trace parser" feature to be one for the vector of line 10, while the others to be zero on the same line. It would be possible that, in a few cases, more than one parser-based features have value one.

With these new features in place, we used again the Naïve Bayes machine learning process and conducted training and evaluation as presented in Section 9.4.2.

Results. Varying the value of the threshold t (see Section 9.4.3), we found the best average results to be at $t = 11$. Table 9.6 shows the confusion matrix on the best results achieved by adding the four parser-based features to the Naïve Bayes approach. It correctly classified 62,093 instances (91.3%), 1,513 more than the previous approach.

Table 9.6: Results adding parser-based features

classified as →		NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	■	31,898	960	97	158	29	0.908	0.962	0.934
Junk	▒	3,087	25,787	325	203	183	0.962	0.872	0.915
Patch	⧅	55	29	1,719	278	1	0.739	0.826	0.780
Source Code	⊠	78	13	185	1,636	2	0.719	0.855	0.781
Stack Trace	▨	9	6	1	0	1,053	0.830	0.985	0.901

Comparing the confusion matrices of the machine learning approaches (Table 9.6 and Table 9.4), we see that the new features helped to decrease the instances wrongly classified as NL text. Being NL the most frequent class (see Table 9.2), it has a strong impact on the evaluation of the MAP hypothesis of Naïve Bayes (see Equation 9.2); since the new features reduced the NL class impact, they play a major role in the classification.

Although achieving the best results so far, this approach has drawbacks. We note that both *patch* and *code* have more than 150 wrongly classified instances: This contradicts the high precision values of the single parsers. It is probably due to the fact that, even if these parser features have a high weight in the computation, they are at the same level of the other features that, being a large number, also influence the results. We expect an approach not having the conditional independence assumption of Naïve Bayes to better model the new features, which are highly inter-dependent. In the following we explore a two-pass classifier approach to better exploit parsers, yet relying on Naïve Bayes qualities.

Unified Classification Approach. To explain our unified classification approach, we refer to Figure 9.5. The idea behind this approach is using Naïve Bayes to evaluate a partial classification only on the features based on *terms*, and then using another machine learning classifier to model the fusion of Naïve Bayes results and parser-based classifications.

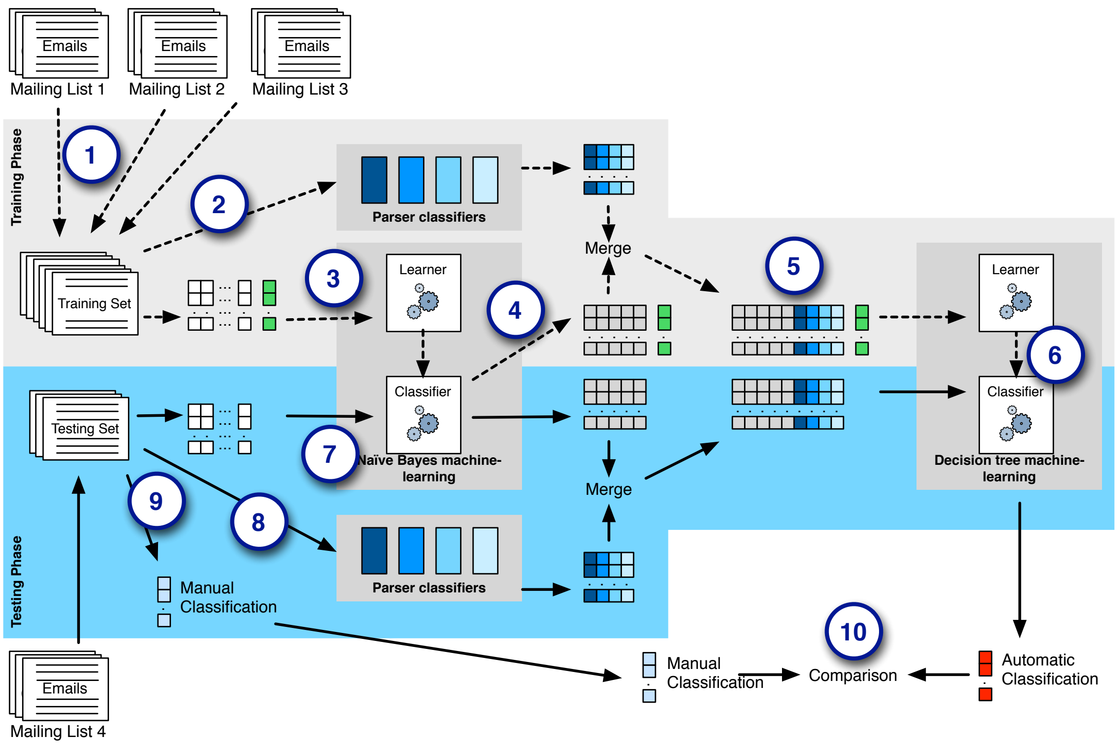


Figure 9.5: Training and Test Process of the Unified Classification Approach

Training. We first (Point 1) extract the emails from the three mailing lists on which we want to train the machine learning algorithms, then we provide them—along with the manual classification—both to the parser-based classifiers (Point 2) and to the Naïve Bayes learning algorithm (Point 3), in the form of feature-vector on words, punctuation, bi-grams, and context. Naïve Bayes trains a classifier, but instead of returning the instance classifications, it outputs a 5-dimension vector for every line: Each dimension represents a class (*e.g.*, *junk*) and the value is the probability of the line belonging to that class, as evaluated by Naïve Bayes. In other words, instead of picking the highest value and providing the final classification, we output all the 5 probabilities and we map them to features, thus reducing the initial features to 5. At the same time, the parsers create other four features, as in the previous approach. Once both feature sets are evaluated, they are merged into a vector of 9 dimensions, plus the manual classification (Point 5). This vector is treated by another machine learning algorithm to train the final classifier (Point 6): The actual output of the training. The choice of the machine learning technique for the second step is critical: We need an algorithm to correctly model the peculiar characteristics of our features. We tried different machine learning approaches, finding that the decision tree [133] is the best suited one, since it is favorable to the parsers' features, which are almost mutually exclusive.

Testing. The test process is depicted in the bottom half of Figure 9.5. We take emails from the fourth mailing list and we remove the manual classification. Then, we provide the emails to the parsers (Point 8) and create the feature-vectors, to be given as an input to the previously trained Naïve Bayes classifier (Point 7). Subsequently, the output of the two technique is merged in a unified 9-dimensions vector, which it is used as input to the second machine learning classifier, previously trained, which outputs the final classification. We compare this classification (Point 10) to the manual one (Point 9) and we evaluate the results. The training and test phases are repeated 4 times rotating the four mailing lists. We tested the approach with a range of *t* values. The highest ratio of correct instances (94.1%) is at *t*=120, which is in the range described in Section 9.4.3. The lowest ratio of correct instances with a *t* value (*i.e.*, 11) within the range is 92.1%; out of the range, values are lower.

Table 9.7: Results of the unified approach on mailing list cross validation

classified as →		NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	■	31,584	1,470	0	87	1	0.937	0.953	0.945
Junk	▒	1,958	27,498	12	115	2	0.943	0.929	0.936
Patch	▧	68	49	1,935	30	0	0.990	0.929	0.959
Source Code	▩	86	118	8	1,702	0	0.880	0.889	0.885
Stack Trace	▨	18	12	0	0	1,039	0.997	0.972	0.984

Results. Table 9.7 shows the results achieved by the approach on the best *t* value. This two-steps approach, which differently merges and model the information, improves the results for all the classes by increasing not only the results related to the parser classifiers (*i.e.*, patch, stack trace, and code), but also those connected to the Naïve Bayes algorithm. The F-measure values are all increased, with a decrease in precision of junk classification and in recall of NL classification, probably due to the overall lower weight given to Naïve Bayes results.

9.5 Threats to Validity

Construct Validity. To classify email content we rely on error-prone human judgment. To alleviate this issue, we devised a web application to ease the annotation process. Two annotators cross-inspected 10% of the emails. They found only 12 erroneously classified lines. We corrected these 12 errors in the set of email that was used for the experiments. Even though we expect the same low error proportion in the rest of the sample, it may affect the accuracy of the results.

Statistical Conclusion. We took representative samples of email populations with a 95% confidence and a 5% error level, which are standard values. Our corpus has 67,792 not empty lines.

External Validity. The approaches we tried may show different results when applied to other software systems and mailing lists. To alleviate this, we chose 4 systems with unrelated characteristics and developed by separate communities. The usage of the mailing list varies, as confirmed by the different line class distributions. To test the generalizability of our approach we conducted cross mailing list validation. A second threat concerning the generalizability is that our approach is tailored to a single object-oriented programming language, *i.e.*, Java. However, since most of the language related line recognition relies on island parsers, it can be easily adapted to other programming languages that have a similar structure (*e.g.*, C, PYTHON), without the need of changing the ground concepts.

9.6 Related Work

Researchers applied natural language analysis techniques to software-related documents and devised approaches to improve the comprehension of the natural language parts. For example, Dekhtyar *et al.* [62] discussed the promises and perils of text mining for natural language software artifacts. Here we focus on research on the recognition of the different parts that compose natural language artifacts.

The work by Bettenburg *et al.* [32] focuses on making the research community aware of the noise in email data and presents the importance of a proper cleaning pre-processing phase. The authors suggest possible filtering heuristics to recognize noise and irrelevant information. Later, Bettenburg *et al.* devised INFOZILLA, a tool to recognize and extract patches, stack traces, source code snippets, and enumerations in the textual descriptions that accompany issue reports [31]. It is composed of four independent filters, one per category, which are used in cascade to process the text. The source code filter exploits an approach inspired by island parsing [136], while the others are based on text matching implemented through regular expressions. In the task of *differentiating documents* (*i.e.*, deciding whether they contain or not each category), INFOZILLA reached almost perfect results, with precision and recall values above 0.95 in all the categories. INFOZILLA has been applied to investigate relevant features of text in issue reports [211].

Development emails differ from bug comments, as the former ones (1) contain a larger natural language vocabulary, since the discussion is not limited to bug related issues; (2) present more noise, generated for example by email headers and authors' signatures; and (3) pose greater challenges in text recognition, since many email clients automatically wrap long lines of text, thus breaking the right formatting [35]. Bird *et al.* proposed an approach to measure the acceptance rate of patches submitted via email in OSS projects [35]. They extracted code patches from emails and used them to analyze the developers' interactions.

Some information retrieval approaches targeted the classification of text or the recognition of information with specific patterns [101], exploiting probabilistic and ML models (*e.g.*, Maximum Entropy Models [25] or Hidden Markov Models [24]). Tang *et al.* addressed the issue of cleaning the email data for text mining [185]. The authors proposed a four-step approach to clean emails: (1) non-natural language text filtering, (2) paragraph recognition, (3) sentence boundaries detection, and (4) word normalization. Their method first filters out email headers, signatures, and program code (*without* a distinction from patches or stack traces); then it recognizes the paragraphs and sentences that compose the remaining natural language text; finally, it corrects misspelled words. The authors randomly chose a total of 5,459 emails from 14 unrelated sources (*e.g.*, newsgroups at Google) and created 14 data sets in which they manually labeled headers, signatures, quotations, and program codes. Given the labelled data, the authors implemented a classifier for each step of their approach. All the classifiers use Support Vector Machines (SVM) and are based on specific features (*e.g.*, number of words). At line level classification, they achieved an f-measure of 0.81 in recognizing code, and 0.98 and 0.90 for header and signature.

Carvalo and Cohen devised methods to recognize signature blocks and reply lines in emails [43]. They worked at the line level and tested the effectiveness of a set of features with many ML classifiers. In the signature detection task the methods reached an f-measure value of 0.97.

In our previous chapter we proposed two approaches based on island parsing to identify, parse, and model the structured fragments in development emails. In this chapter we embed them within machine-learning techniques. Machine-learning helps us to avoid the cost of hand-coding all the classification rules and cover unexpected cases, thus obtaining a more robust classification approach. We strived for an approach with a fine granularity and a wide breadth, to provide a robust classification that can be used to increase the quality of subsequent analyses.

9.7 Summary

Email communication contains valuable information to support software development, comprehension, and analysis. In this chapter, we presented a technique to automate the analysis of such valuable, but also voluminous, data that is specifically tailored for software engineering.

We presented a unified 2-step approach that fuses supervised machine learning approaches with island parsing to perform automatic classification of the content of development emails into five language categories: natural language text, source code, stack traces, code patches, and junk. The results obtained are very positive, even with cross mailing list validation. In fact, parser-based classifiers are mailing list independent and offer a solid basis made more robust by the probabilistic machine learning approach.

Reflection. This work is a step toward a more effective exploitation of email data. Although it does not prove our thesis by itself, it is a valuable mining approach for unstructured data that allows improved traceability recovery techniques, refined artifact summarization approaches, and more precise fact extraction methods. These are all activities that have been proved to support software understanding and development.

Part IV

Epilogue

In Part II and Part III, we aimed at proving our thesis by creating approaches to overcome the challenges in mining unstructured software data, and then by building on top of these approaches to devise analyses and tools to support software development and program comprehension. In the following (and last) part of this dissertation, we take a step back from our approaches and findings, and we conclude our work.

In this dissertation, we particularly focused on development mailing list, especially because of their noisy and mixed language content that constitutes a veritable acid test for our mining approaches. Moreover, we chose OSS development mailing lists for our analyses, because they have been considered—historically—the hub of project communication at the inception of the first OSS communities, such as Linux and Apache. For this reason, when studying OSS developers' communication, many researchers focused on development mailing lists: For example, to investigate the handling of patches [35; 165], or developers' social networks [36]. We followed the same path.

As a first step in this epilogue, in Chapter 10, we re-investigate the role of OSS development mailing lists, which have been the subject in most of our dissertation. In fact, during our analyses we sensed that mailing lists in OSS communities have been facing a shift with respect to their usage in the first years of the first OSS projects. Future work should be based on informed assumptions, thus we decided, before concluding, to conduct a work to update our knowledge on the mailing list data available nowadays and probably in the future. This helps us to target our future research to the most important information available in mailing list and also to the most promising repositories of unstructured data.

In Chapter 11, we conclude this work, by enumerating the contributions we made in the course of our dissertation and by describing steps we envision as valid future work.

Chapter 10

Communication in OSS Mailing Lists

Open source software (OSS) development teams use electronic means, such as emails, instant messaging, or forums, to communicate. Conversations in OSS settings are typically conducted in an open public manner and are stored for later reference [36]. For this reason, OSS communication repositories offer a rich source of historical information, which can be used, for example, to observe software processes [175], to understand software developers' communication dynamics [172], and to improve development practices [178].

Mailing lists have been considered—historically—the hub of project communication at the inception of the first OSS communities, such as Linux and Apache. For this reason, when studying OSS developers' communication, many researchers focused on development mailing lists: For example, to investigate the handling of patches [35; 165], traceability concerns [18], or developers' social networks [36].

These and other studies (*e.g.*, [146; 38; 154; 171; 12; 168]) are mostly based on the conventional wisdom that the role and usage of the development mailing lists (of the analyzed project) are similar to that of Linux [162] or Apache [134] in their first years. This leads to a number of assumptions, such as that development mailing list "*are primarily concerned with the software under development*" [154], and that "*communications by means of* [e]*mail is the only possible way for* [OSS developers] *to interact with each other.*" [23]

Nevertheless, there is no clear, updated, and well-rounded picture of the communication taking place in open source development mailing lists that supports these assumptions. In fact, at our disposal, we only have either abstract and outdated knowledge (*e.g.*, obtained as a side effect of the analysis of the Linux project), which does not consider the recent shift of interest to new social platforms (*e.g.*, GitHub and Jira), or a very specialized understanding (*e.g.*, regarding specific information, such as the process of code review [168]), which does not take into account all the information that can be distilled from development emails.

In the previous parts of this thesis, we (mostly quantitatively) analyzed a large number of emails, and we sensed a change in the usage of development mailing lists across OSS systems. In this chapter[1], we present an analysis to update our knowledge on development mailing lists, so that we can better guide future work on mining this source of information.

1 the work presented in this chapter was performed in collaboration with Anja Guzzi, Delft University of Technology.

10.1 Overview

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

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

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

Contributions of the chapter. In this chapter, we present the following contributions:

- *We conduct a qualitative study to understand what data can be found in OSS mailing lists*. We conduct this research to guide future work on this form of unstructured software data.
- *We create a coding system that is reusable for analysis of developer communication in general, and mailing lists in particular*.
- *We assess the frequency of discussion topics in development mailing list*. In particular, we found that implementation details are not extremely prominent.
- *We create two benchmarks: one for email thread categorization and one for resolving aliases of participants.* Our benchmark are comprised of more than 500 manually classified threads and more than 310 resolved aliases and email addresses.

Based on our findings, we analyze and discuss the implications for researchers and practitioners (Section 10.10).

Structure of the chapter. In Section 10.2 we present the methodology we followed in this work, also presenting the research questions. In Section 10.3 we derive the topics discussed in the development mailing list of LUCENE. In Section 10.4 we study the distribution of discussion topics. In Section 10.5 we study the participants of the mailing list, in particular the developers taking place in the discussions. In Section 10.6 we discuss the current role of development mailing list in the analyzed project. In Section 10.7, based on our findings, we analyze and discuss the implications for researchers and practitioners. We present the limitations in Section 10.8 and the related work in Section 10.9. Section 10.10 summarizes the chapter.

10.2 Methodology

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

We chose LUCENE for the following reasons: (1) LUCENE is a mature project with a large user base and an established community of developers. (2) It was started in 1999 by a single developer, who initially guided it as a "benevolent dictator". In 2001, LUCENE joined the Apache Software Foundation and became a *foundation*, with a well-organized, hierarchical governance structure and formalized policies[2]. (3) The previous work describing the communication occurring in the development mailing list of OSS projects (*e.g.*, [162; 112]) dates back to the early 2000s, it is high-level and focuses on Linux, which is more of an exception than the rule in OSS projects [162]. LUCENE's organizational structure sets it apart from the benevolent dictatorship of Linux; by choosing LUCENE we aim at having an updated knowledge of contemporary developers' communication in the development mailing list in a more common OSS setting. (4) LUCENE has a *publicly* archived development mailing list with a *declared* intent: The developer discussion `dev@lucene` list is " *where participating developers of the Java Lucene project meet and discuss issues concerning Lucene* [...] *internals, code changes/additions, etc.*"[3]

10.2.1 Research Questions

Our investigation revolves around four research questions:

RQ1: What topics are development mailing list participants talking about?

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

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

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

10.2.2 Research Method

We followed the approach depicted in Figure 10.1: (**1**) we modeled all the mailing list emails, (**2**) we reconstructed threads of discussion (removing auto-generated ones), and (**3**) we randomly extracted 1,000 threads. Using open card sort [20] (see Section 10.2.4), we manually analyzed the threads and extracted categories of discussion (**4**). To ensure the integrity of the extracted categories, we sorted threads several times and iteratively refined the catalogue (**5**). During the cart sort, we took notes about the mailing list, its role, and the communication occurring in it (**6**). We validated the resulting catalogue of categories using closed card sort (**7**). We complemented the automatically collected email data by resolving aliases and by adding information about which participants were developers (**8**). Finally, we conducted a statistical analysis on the obtained categorized threads (**9**).

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

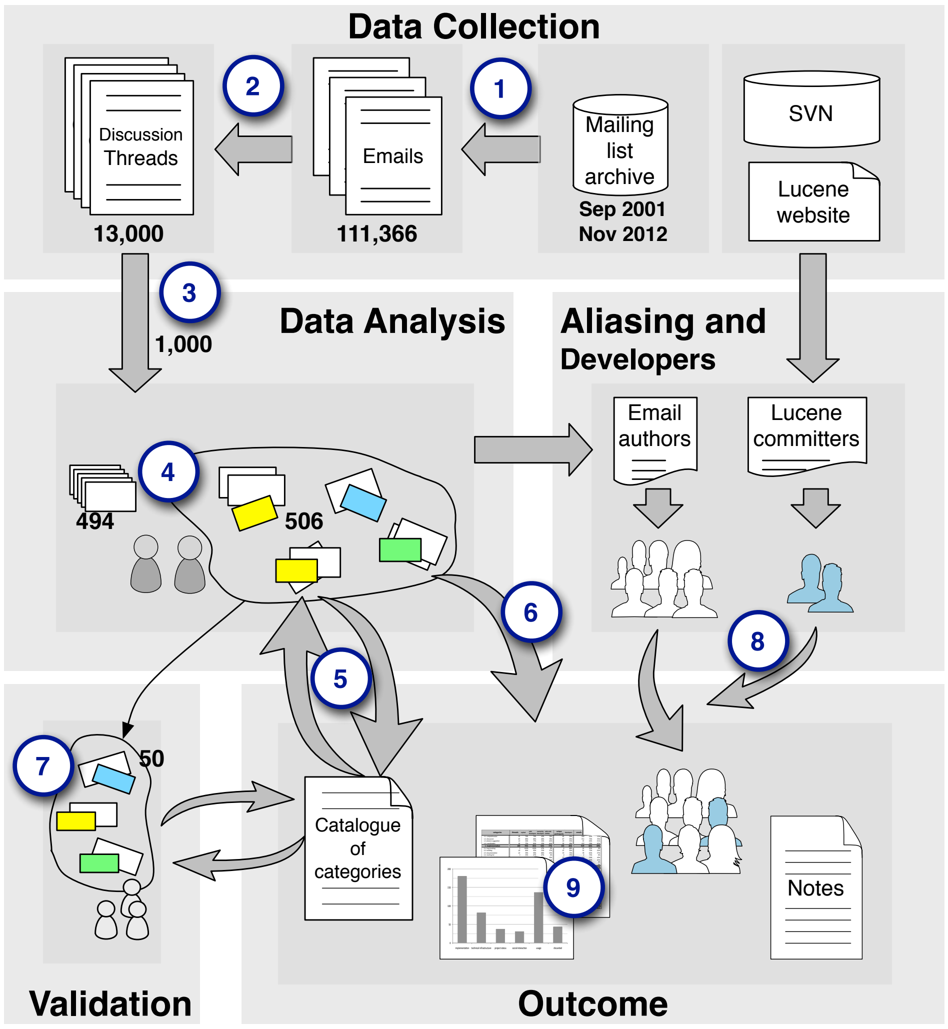


Figure 10.1: The mixed approach research method applied.

10.2.3 Data Collection

We collected the necessary data using the MBox importer in Miler (see Section 3.2.1). We also processed the data in the following way:

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

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

10.2.4 Card Sort

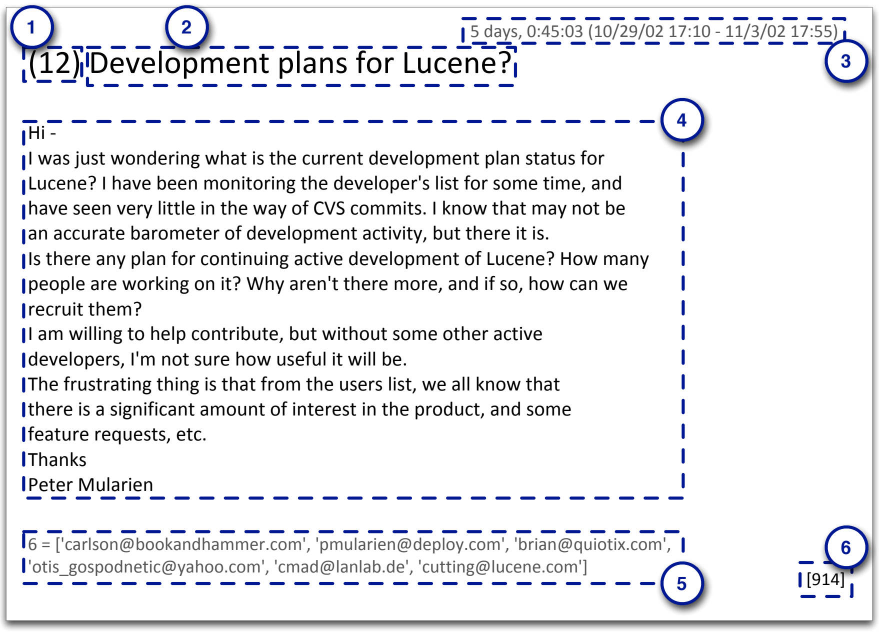
To group the email threads we used *card sort*, a technique used in information architecture to create mental models and derive taxonomies from input data [20]. We used it to organize the threads into groups to abstract and describe mailing list communication. A card sort has 3 steps: (1) *preparation* (select card sort participants and create the cards); (2) *execution* (sort cards into meaningful groups); and (3) *analysis* (form abstract hierarchies to deduce general categories).

Preparation. We created all cards from the sample resulted from the data collection. Each card (exemplified in Figure 10.2) represents a thread and includes: (1) number of emails, (2) subject, (3) duration, with timestamp of the first and last emails, (4) the first 15 lines (removing white lines) in the body of the initial email, (5) email addresses of the participants involved, and (6) an univocal `id` for later reference.

Execution phase. The first two authors analyzed the cards applying *open* (*i.e.*, without predefined groups, as they emerge and evolve during the sorting process) card sort, adapting the guidelines for the analysis of qualitative data with grounded theory [81]: They avoided information related to Lucene (*e.g.*, its website) and the literature closely related to mailing list communication, as this could have sensitized them to look for concepts related to existing theory, thus hindering innovation in organizing the threads. They often interrupted the card sorting to memo an idea or concept potentially useful for later analysis (see Section 10.6). When necessary they consulted the entire thread online. Since the rigor of the card sorting method is in its analysis [129], instead of working separately on different cards, and checking the consistency of the sorting and merging the cards in a later phase, they used *pair-sorting*. This requires significantly more time, but it brings more value to the analysis as they discussed discrepancies in their thoughts for each card during the card sorting itself.

Analysis phase. To ensure the integrity of the emerging categories, the first two authors did a second pass on all the analyzed cards, starting from small groups that could not be included in any larger group, and re-categorizing these cards by redefining some categories. Subsequently, they analyzed the remaining cards to completely describe the catalogue of thread categories (see Section 10.3).

We conducted a validation to verify whether the catalogue was written in a clear and understandable way that was capturing all the facets of each category (see Section 10.8).



(12) Development plans for Lucene?

5 days, 0:45:03 (10/29/02 17:10 - 11/3/02 17:55)

Hi -
I was just wondering what is the current development plan status for Lucene? I have been monitoring the developer's list for some time, and have seen very little in the way of CVS commits. I know that may not be an accurate barometer of development activity, but there it is.
Is there any plan for continuing active development of Lucene? How many people are working on it? Why aren't there more, and if so, how can we recruit them?
I am willing to help contribute, but without some other active developers, I'm not sure how useful it will be.
The frustrating thing is that from the users list, we all know that there is a significant amount of interest in the product, and some feature requests, etc.
Thanks
Peter Mularien

6 = ['carlson@bookandhammer.com', 'pmularien@deploy.com', 'brian@quiotix.com', 'otis_gospodnetic@yahoo.com', 'cmad@lanlab.de', 'cutting@lucene.com']

[914]

Figure 10.2: Card Sort: Example Card

10.2.5 Aliasing and Identification of Developers

Resolving multiple identities (aliases) is fundamental to prepare mailing list data for the statistical analysis of the participants [38]. Although a number of approaches to solve aliasing have been proposed (*e.g.*, [36; 87]), this task cannot be fully automatized. To avoid bias in our statistical results, we manually resolved aliasing in our data. We started by aggregating on email addresses, to resolve cases with multiple author names. Then, we manually inspected all possible combinations of names and email addresses. One challenge we encountered regards a handful of participants using distinct names and addresses (*e.g.*,'John: `johns@address1.com`', and 'spacej: `spacej@address2.co.uk`'). To resolve these cases, we read the emails sent from these addresses. To answer the research question regarding developers communicating in the mailing list participants (*i.e.*, RQ3), we also identified the official committers of the project: We matched names and addresses in our sample with the official list of committers[4]. We also extracted developer user names from the versioning system log. Matching developers was time-consuming, as only few developers use their *[user-name]*`@apache.com` address listed on the LUCENE website.

10.3 What are mailing list participants talking about?

We extracted email data from the LUCENE development mailing list,[5] from its inception (Sep 2001) to Nov 2012, totaling 111,366 emails. We aggregated them into threads and removed automatically generated messages. From the resulting 13,019 discussion threads we randomly sampled

4 `http://lucene.apache.org/whoweare.html`
5 `org.apache.lucene.java-dev`

1,000 threads and printed the corresponding cards for the card sort. After sorting the first ca. 300 cards, the new threads started merging in the same groups, reaching a saturation effect [81]. To add confidence that the saturation point was reached, and to improve the significance of the subsequent statistical analysis, another 200 cards were sorted, reaching a sample of 506 threads. The remaining cards were discarded.

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

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

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

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

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

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

Technical Infrastructure. Most OSS software projects rely on a technical infrastructure to support development, maintenance, and the building process, and to facilitate the communication among project contributors [73]. This category covers email threads related to such an infrastructure; the topics of discussions are (B.1) BUILDING SYSTEM (*e.g.*, notification of problems with the building system), (B.2) DOCUMENTATION (*e.g.*, decisions on the javadoc), (B.3) ISSUE TRACKING (*e.g.*, move to a new tracking system), (B.4) MAILING LISTS (not only the development mailing list itself, but also *e.g.*, the user mailing list), (B.5) PROGRAMMING LANGUAGE (*e.g.*, the version of [programming language] to use), (B.6) TESTING (*e.g.*, how to use the continuous testing system), (B.7) VERSIONING (*e.g.*, discussions on branches), and (B.8) WEBSITE (*e.g.*, threads on what content to put in the website). Authors of infrastructure threads write to the list for different reasons, such as sending notifications, discussing problems, and posing questions.

Project Status. As described in previous work (*e.g.*, [162; 134; 38]), development mailing lists are also used to raise awareness on the status of the project and to discuss future steps. This category regards these kind of topics, and includes two groups of threads: those about (C.1) PLANNING the future development of the project, and those about (C.2) RELEASES. Authors of PROJECT STATUS threads write to the mailing list to announce a new release, to decide which issues to fix for a milestone, or to discuss the ongoing activity on the project.

Social Interactions. Socializing is essential for the long-term survival of OSS projects [64], and mailing lists play an important role in this context [73]. Participants write to the mailing list about the norms, values, and perspectives that are part of the community's operational structure, and to coordinate with others. This category revolves around these social interactions, and threads are about (D.1) ACKNOWLEDGEMENT of efforts (*e.g.*, replying to a code commit to thank the author), (D.2) COORDINATION (*e.g.*, raising awareness about an issue in the bug repository, or notifying a participant's absence), greetings and suggestions to (D.3) NEW CONTRIBUTORS, and (D.4) SOCIAL NORMs governing the behavior of mailing list participants (*e.g.*, advices on successfully submitting a patch). Authors of such threads notify their absence, welcome new members, thank someone for a well-done bug fix, and tell everyone about newly submitted issues.

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

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

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

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

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

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

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

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

10.4 How often do participants talk about each topic?

Our second research question seeks to understand how much participants discuss each topic and how prominent are implementation details.

Figure 10.3 shows the distribution of the threads among the different categories (see also column 'threads' in Table 10.1).

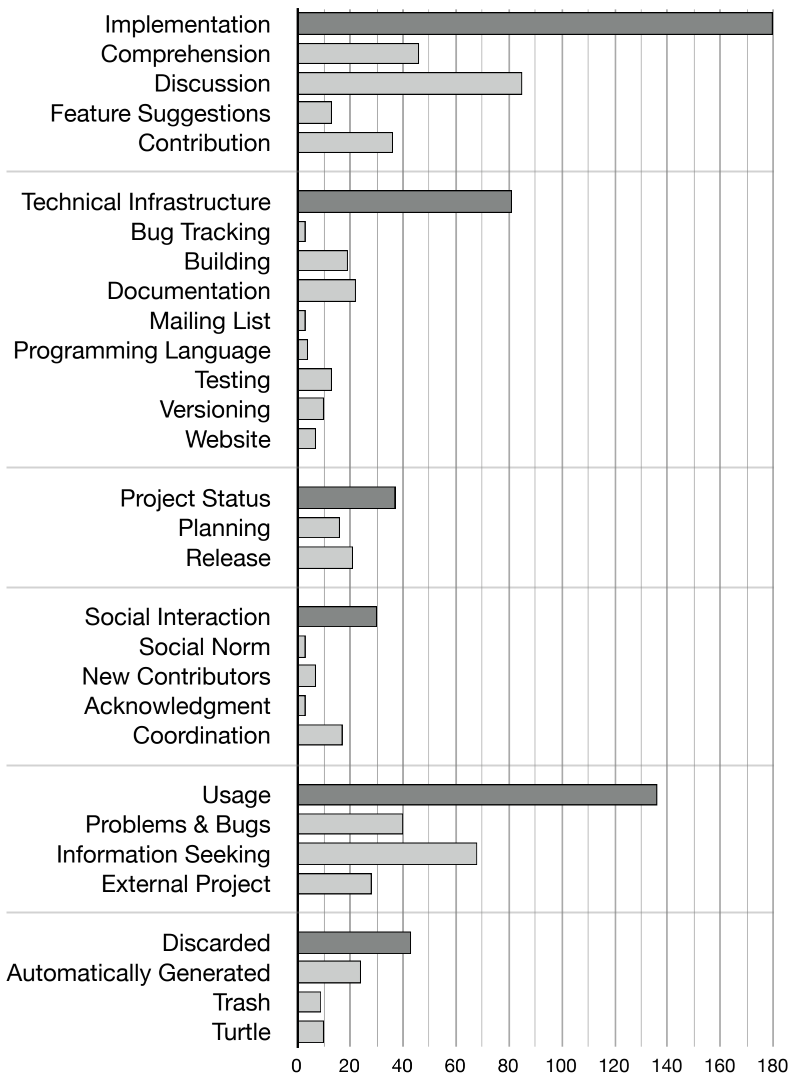


Figure 10.3: Distribution of threads per category.

Implementation is the most frequently occurring category, comprising 36% of the threads. Since the declared aim of the *development* mailing list of Lucene is to be where "*participating developers [...] meet and discuss issues concerning* Lucene *[...] internals, code changes/additions,* etc.", we were surprised that—in reality—IMPLEMENTATION threads count for just a bit more than a third of all the threads. This changes from the Linux kernel mailing list (often used for studying developers' interaction), where IMPLEMENTATION threads *"form the large majority of the traffic on the list."* [85]

In comparison, we found the ratio of USAGE threads in the mailing list to be surprisingly high (27%). In particular, half of these threads regard INFORMATION SEEKING (13% overall), in spite of a note on the LUCENE website exhorting participants to *"not send mail to this list with usage questions or configuration questions and problems"*. Moreover, threads regarding PROBLEMS AND BUGS account for 8%. Even considering sampling limitations (see Section 10.8), in LUCENE these threads would corresponds to less than half of the bugs reported in JIRA (up to one fifth, when considering other types of issues), meaning that in LUCENE the mailing list may not be the primary channel for discussing and reporting problems and bugs.

Threads regarding TECHNICAL INFRASTRUCTURE total 16%. The less frequent thread categories are SOCIAL INTERACTION and PROJECT STATUS. It was surprising to us that, despite the mailing list always having been considered the hub for OSS project communication [162; 73], only 7% of the threads in our sample regard the project status, and 6% regard social interactions among participants.

Finally, there is a not negligible portion (8%) of threads DISCARDED during the card sorting. Besides 10 threads with no clear meaning (TURTLE), and—despite the fact that we performed an rigorous pre-processing and data cleaning phase (Section 10.2)—a substantial amount of noise (7% of the total threads, from AUTOMATICALLY GENERATED and TRASH threads) was still present our sample. We also notice that these threads cannot be clearly distinguished from the other categories: a third of them are replied (*e.g.*, there are threads automatically generated from the wiki, which all have the same subject and thus get threaded), almost a third include developers in the emails (*e.g.*, svn commits initially sent to the mailing list results as sent by the developer author of the commit), and finally almost a fourth of these threads contain code (*e.g.*, svn commits, and change logs from the wiki pages).

10.4.1 How prominent are implementation details?

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

In previous work, Bird *et al.* reported that the mailing list is made of more than implementation. They distinguish between *process* and *product*, and use the presence of source code names, such as class names, as classifiers: *"Messages that include these source code names are classified as product and the rest are classified as process"* [38]. We also apply this distinction to our data and verify whether and how it fits to our categories. We considered the entities mentioned in all the releases of LUCENE, and we analyzed threads to determine whether they contained code entities. Results from our analysis can be seen in the column 'with code entities'.

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

To verify to which degree Bird *et al.*'s classification fits to our data, we first need to define which of our own categories are part of *product*. According to the description, these would correspond to our IMPLEMENTATION category alone. However, USAGE and DISCARDED threads would not fit in either definition: we decided to include USAGE as *product* (since LUCENE is an API, many usage questions regards its code artifacts), while we consider DISCARDED as *process*.

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

Finally, we used the Fisher's exact test [193] to test whether there is an association between referencing a code entity and the two categories product and process. The test resulted in a p-value of < 0.001 indicating the existence of such an association. Next, we investigated the strength of this association by computing the odds ratio (OR) [193]. OR measures the probability of a thread referencing at least one code entity to be categorized as product compared to a thread not referencing any code entity. We obtained an odds ratio of 4.082 with a lower and upper confidence interval of 2.781 and 6.038. Consequently this probability is four times higher, thus confirming the association between referencing a code entity and the categorization of threads into product or process.

10.5 Is the development mailing list only for developers?

Once our categories were stable, and after performing several card sort iterations to ensure the integrity of our categories, we resolved aliasing and determined which participants were project developers (*i.e.*, those with commit privileges). Table 10.1 shows the statistical information we collected on the sample of threads categorized in the card sorting process. We include email granularity for completeness.

10.5.1 What do developers focus on?

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

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

Furthermore, we notice that only 21% of the USAGE threads were started by a developer, and, in particular, only 4% of the INFORMATION SEEKING threads. It is not very surprising that these threads are not started by LUCENE developers. However, developers also start EXTERNAL PROJECT

Table 10.1: Categorization of email threads.

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

threads: They often have side projects, built on top of Lucene, they want to mention in the mailing list (*e.g.*, announcing a new release).

10.5.2 Dynamics of interactions

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

The column 'Unique participants' indicates the number of individual people participating to discussions threads. When the number of participants is lower than the number of threads, this means that people are participating in more than one thread (*e.g.*, this is the case in the imple-

MENTATION category). Similarly, a higher number of participants than the number of discussion threads indicates "one-timers" (we observe this in the USAGE and PROJECT STATUS categories).

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

10.5.3 The overall picture

Figure 10.4 puts all the threads in our sample in a nutshell: It shows, by category, how many threads are with *vs* without code entities (left vs right side), with *vs* without participating developers (top vs bottom bar), and how many of the latter have been initiated by a developer (light vs dark color). The exact amount of threads is reported for each "type".

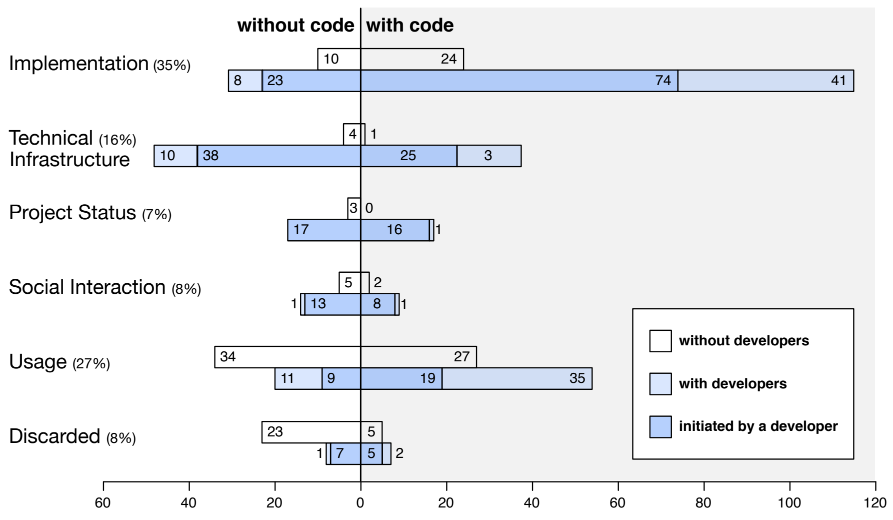


Figure 10.4: Distribution of the types of threads.

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

10.6 What is the role of the development mailing list?

By answering the previous research questions, we found that the official description of the aim of the development mailing list does not correspond to its real usage. Our fourth research question seeks to understand the role of development mailing lists for the communication in OSS at large. We attempt to achieve this by triangulating the information that we obtained by reading the 506 threads during the card sort, by analyzing the statistical data on the categories, and by searching more facts in the rest of the mailing list.

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

Previous literature stated that mailing lists are "*the bread and butter of project communications*" [73], and in particular that "*the developer mailing list is the primary communication channel for an OSS project*" [85]. Reading the analyzed emails, however, makes it clear that the development mailing list is just *one* of the communication channels used in a OSS project; in fact, other channels also play an important role:

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

IRC: Participants talk about the project and implementation details also on the development IRC channel (created in Apr 2010): "*I propose that we chat on irc at #lucene-dev* [...]. *I'd like to discuss the core elements of the Spatial Strategy API, namely makeQuery,* [...], *and SpatialOperation.*" The channel was created in Apr 2010, and the project currently keeps a log[6] of it.

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

In person: We found evidence that developers also have a number of in person meetings where they discuss on project details (*e.g.*, "[Developer] *and I talked a little bit about this at the ApacheCon*"). This is in line with the findings of Aranda and Venolia concerning software defects. They found that "histories of even simple bugs [...] cannot be solely extracted through the automated analysis of software repositories." [5]

10.6.2 Is the mailing list for driving coordination?

Previous work reported that a portion of the communication taking place in the mailing list regards coordination between developers as they work together on the software [162; 38]. Surprisingly we found a very small amount (3%) of COORDINATION threads, with an average of less

6 `http://colabti.org/irclogger/irclogger_logs/lucene-dev`

than two emails per thread; moreover, most of these threads were not for fostering collaboration on the implementation, but for raising awareness on already accomplished work.

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

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

10.6.3 Is the mailing list used for peer code review?

Rigby *et al.* reported that OSS mailing lists are also used for submitting patches and performing peer code reviews [168]. We indeed found that most patches led to a purely technical discussion, while some others also led to a discussion of project objectives, scope, or politics.

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

10.6.4 Is the mailing list the hub of project communication?

Although other researchers also found that the development mailing list is not the only channel of communication in OSS projects (*e.g.*, [38; 171]), it has always been considered the *hub* of project communication. For example, Mockus *et al.* reported that developers use *"email lists exclusively to communicate with each other"* and that *"due to some annoying characteristics of the* [issue tracking system], *very few developers keep an active eye on* [it]." [134].

When more communication repositories exist, the policy of most OSS projects is to transfer all the official decisions and useful discussions to the mailing list [73], so that they can be later retrieved. These traceability links between the development mailing list and other communication repositories must be manually created and updated. We found some cases in which the traceability link was established, but, more often and in line with the findings of Sarma *et al.* [171], we found a clear *disconnection* among repositories, which led to coordination issues and duplicated/lost information. For example, because of multiple communication repositories, developers need to raise inter-repository awareness (*e.g.*, "*I submitted a patch for* [Jira issue] *a month ago,* [...] *it hasn't been picked by anybody yet*"), ask where a discussion takes place (*e.g.*, "*were there emails about it or it has been discussed on IRC?*"), and go back and forth between the same discussion taking place in more venues (*e.g.*, "*We would like to implement* [this], *which was discussed in Jira*"). Overall, our investigation provides evidence that the various communication channels work in parallel, remain disconnected between one another, and the development mailing list does not play (anymore) the role of a hub.

10.7 Implications

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

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

Automatically recovering traceability links among communication repositories would free developers from the task of recovering scattered traces of previous communication, and would help researchers having a more complete picture of the development process. More tools for maintaining awareness would be also necessary to improve developers' productivity. Since the advent of better issue tracking systems led to a shift in the habits of OSS participants toward different communication means, we should investigate the features in issue tracking systems that produced this change of direction. As a long-term vision, we could consider the creation of Integrated Evolution Environments (IEE) that would aggregate all the facets of software development into a single comprehensive scenario; this in opposition to current Integrated Development Environments (IDE) that offer a partial view by focusing only on the source code.

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

On Software Development. Not only committers respond to the development mailing list, but also other people are very active. We could consider techniques for finding code experts not only among active contributors, but also among active respondents of the mailing list. Moreover, considering the shift to other communication repositories, mailing list may not be the right venue for studying code review anymore. In this context we can further investigate the role of issue tracking systems and social coding websites such as GitHub.

10.8 Limitations

One potential criticism is that a case study with one project may provide little value. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science [71]. To understand mailing list communication we read emails spanning 11 years of mailing list usage, and written by 155 diverse participants. To answer our research questions, we also analyzed data from the code repository, the project website, and email threads external to our sample. Nevertheless, our study needs to be replicated to reach more generalizable conclusions to the posed research questions.

To ensure that the thread categories emerged from the card sort were clear and accurate, and to judge whether our set of category provides an exhaustive and effective way to organize mailing list communication, we conducted a validation phase that involved three people external to the pair-card sort. Three software engineering researchers conducted a closed card sort on 50 cards (10%) randomly selected from our sample. They observed that the 6 main categories were clear and covered all thread topics. We measured inter-rater agreement: The Fleiss' Kappa value for the four ratings of the random sample was 0.657 (*i.e.*, substantial agreement) for the six categories, and 0.505 (*i.e.*, moderate agreement) for the 24 sub-categories (which were more difficult to be all recalled by participants). To verify whether there was a systematic error in our catalogue, we also measured the inter-rater agreement among the three experiment participants. Their agreement was 0.592 for the main categories, and 0.458 for sub-categories (both corresponding to a moderate agreement, suggesting there was no systematic misinterpretation).

Threats to validity—Concerning *internal* threats, the sample size (506) of threads provides a 98% confidence level and 5% error on subsequent estimations of proportions [193]. Concerning *external* threats, other OSS projects use communication tools similar to LUCENE, for example, 87 other Apache projects are also using the JIRA issue tracking system and have IRC channels. However, team dynamics may differ and our research should be repeated in other contexts.

10.9 Related Work

By analyzing OSS development mailing lists, researchers provided insight in social aspects of software development. For example, researchers exploited email *metadata* (*e.g.*, author, date, and time) to conduct quantitative social analyses: Bird *et al.* proposed techniques to mine email social networks [36], and investigated social interactions in OSS projects [38]; Ogawa *et al.* visualized social interaction among participants in OSS projects [146]; Tang *et al.* proposed techniques for identifying the country of origin of participants in OSS mailing lists, and conducted a geographic analysis [186]; and Shihab *et al.* showed that mailing list activity is related to source code activity [178]. Researchers also quantitatively analyzed the *text* of emails: Pattison *et al.* studied the frequency with which terms of software entities are mentioned in emails, and correlating it with the number of system changes [154]; and Baysal and Malton searched for a correlation between discussions and software releases [23].

Most of the aforementioned work is quantitative and based on the premise that development mailing list communication mostly regards the implementation of source code artifacts. This assumption derives from the knowledge about OSS systems provided by seminal literature such as "The Cathedral and The Bazaar" [162]. Few studies analyzed the *content* of OSS mailing list communications and mostly focused on specific traits of the communication. Gutwin *et al.*

read mailing list archives to study group awareness in distributed development [85]. Rigby *et al.* analyzed mailing lists to study the OSS code reviewing process (*e.g.*, [168]). Mockus *et al.* studied the Apache Server development process finding that the mailing list play a central role for communication, coordination, and awareness [134]. We wanted to obtain a comprehensive knowledge of communication in development mailing lists of OSS projects.

Our work is also related to data quality: By knowing what data is available in mailing list repositories, we can devise better techniques for extracting relevant, unbiased, and comprehensible information. In this vein, researchers have studied bug repositories [211] and code repositories [102] to understand what information is more relevant. They also analyzed the impact of data quality on mining approaches and analyses (*e.g.*, [187]). In the context of mailing list data, Bettenburg *et al.* showed the risks of using email data without a proper cleaning pre-processing phase [32].

10.10 Summary

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

Reflection. The findings of this chapter show that the usage and content development mailing lists in OSS projects could be different from what we might assume from previous work. Although the results we found in previous chapters are not invalidated by the findings of this chapter, we deem that it is appropriate to take this change of communication trend in consideration, especially when deciding which unstructured data repositories to analyze. In fact, mailing lists might be not the best source of information about a software system and might be not optimal to study certain software development processes. The techniques we presented in the Part II and Part III can also be used in the case that the unstructured repositories to be mined are not development mailing lists.

Chapter 11

Conclusion

In this dissertation we proposed our thesis, which asserts that by mining the email data produced during the evolution of a software project, we are able to disclose a new source of information that can be used to understand and support software development.

We presented the motivation behind our work, we explained the approach we followed, and we presented the challenges we faced. To validate our thesis, we focused on email data and we took two interconnected directions: (1) We used the unstructured and qualitative information in mailing lists to *enrich* models obtained through the analysis of structured data, and (2) we considered mailing lists as an *alternative* data source that give additional and complementary information, which is independent from other forms of data.

In the former direction, we developed methods for restoring traceability links among emails and classes, we devised a new set of "popularity" metrics, and we included email information in the IDE. In the latter direction, we tackled the issue of extracting the structured data embedded in the natural language of emails. We reached interesting *classification* results, by using lexical methods, currently, we are also implementing more sophisticated techniques based on island parsing that also allows us to *understand* and *expose* the meaning of extract structured information from emails. We showed that our mining techniques are valuable for supporting software understanding and development.

11.1 Contributions

During the course of this dissertation, we made a series of contributions to the state of the art in mining unstructured data to support software understanding and development. We summarize the major ones in the following.

11.1.1 Exploratory Investigation

To mine unstructured data, researchers have been experimenting with technologies adopted from related research fields. Techniques such as topic models from Information Retrieval, hierarchical clustering from Data Mining, or Natural Language Processing have been proven limited, in a sense that they are often laboriously tailored to the intricacies of the underlying data and intended use cases. As a result, a plethora of hand-crafted techniques emerged and have been proposed to mine unstructured data. The ad-hoc nature and terse documentation of these techniques, however, hinder their use for other tasks: this variety makes it hard for researchers and practitioners to determine the appropriate technique(s) to deal with the problem at hand and ways to use the selected technique effectively. Moreover the challenges of using unstructured data were not clear, since too much scattered across different analyses.

Therefore, we decided to focus especially on one form of unstructured data, *i.e.*, development emails, and carry out our research in an iterative explorative fashion, to find the challenges that we had to face to prove our thesis that the content of unstructured data is a valuable information source to support software engineering activities. During our investigation, and by studying the work in the area of mining software repositories (see Chapter 2) we found two challenges in exploiting email data for software engineering: (1) disconnection between emails and code artifacts, and (2) the noisy and mixed-language nature of email content. .

In Section 3.2, we introduced our toolset Miler, which we progressively and iteratively devised to import, process, store, and analyze both email and source code data, during our research. Among the features of Miler we have: (1) tools to import and model email and code data according to a meta-model we devised; (2) a transparent storage of imported data into a database for persistence; (3) an extensible web application for visually interacting with imported emails; (4) tools for recovering traceability links, extracting metrics, detecting structured fragments; and (5) a framework for island parsing.

11.1.2 Email Data and Source Code Artifacts Reconnected

In the second part of the dissertation, we presented the importance of reconnecting email data to source code artifacts for using the former to support software understanding and development. We presented a comparison of techniques for recovering the traceability links between emails and source code artifacts, and then we presented two analyses aimed at using this information in software analyses and development.

Recovered the traceability links between emails and source code. In Chapter 4, we presented different lightweight approaches we devised that, by exploiting the nature of emails and naming conventions of software artifacts, are capable of establishing a bi-directional link between source code entities and emails. Our approaches do not require pre-computation (which usually is time-expensive and compromises dynamism and interactivity of applications), but can be directly used at run-time, for example to catalogue an email with its references, or to check for the references to one class in all the e-mails archive. Implementing these techniques is not enough: One needs to be sure they perform correctly. With a manually created benchmark, we showed that approaches can reach significant results in terms of accuracy. This indicates that heavyweight techniques may be not necessary to achieve good results in finding the traceability links between source code and emails. We confirmed this hypothesis in the second part of the chapter were we compared our techniques to more sophisticated IR methods (*i.e.*, vector space model and latent semantic indexing), which have proved to be effective in other traceability tasks.

Improved defect prediction with email data. In Chapter 5, thanks to the recovered traceability links between emails and source code, we presented a first set of new metrics we devised to enrich a system model with information extracted from email archives. Such metrics seize the "popularity" of source code entities in the discussions taking place in emails. In mailing lists, the entities that are discussed are not only the most relevant for the development, but also the most exploited during the software usage. Moreover, the email content is expressed using natural language, which does not require the writer to carefully explain all the abstractions using the same level of importance, but permits to generalize or focus on specific concepts. We used our new popularity metrics to perform defect prediction for object-oriented systems at class level, and we compared their predictive power to that of metrics obtained through structured data (*i.e.*, object-oriented metrics, change/defect metrics). We achieved results similar to source code metrics, but inferior to change metrics. However, the most interesting contribution of our metrics is that, combining the metrics extracted from repositories with different form of data, we can improve the overall predictive power (more than 15% on average). This shows that the data we find in development email archives provides new information that is orthogonal to that offered by structured data sources, such as source code or SCM systems.

Integrated Email Communication in the IDE. In Chapter 6, thanks again to the recovered traceability links between emails and source code, we presented REmail, an Eclipse plugin to integrate email communication in the IDE. REmail recommends the emails that are related to specified code artifacts, by using the aforementioned lightweight linking techniques we devised. This reduces the amount of messages to be read by orders of magnitude, and lets practitioners focus on the emails related to their tasks. In addition, REmail is a modular plugin for the Eclipse IDE, thus among other benefits, it allows developers to (1) simultaneously inspect code and content of messages, easily (2) prompt recovered traceability links between code and emails, and (3) minimize the disruptive context switches necessary to access email data while programming. In the second part of the chapter, we studied two OSS systems by using REmail and we found that the email information, as displayed by REmail, helps to find entry points in an unknown system, understand software evolution, identify experts, and complement missing documentation.

11.1.3 Unstructured Data Restructured

In the third part of the dissertation, we presented the importance of giving a structure to the content of development emails, in order to extract relevant, meaningful, and contextualized information. We presented three techniques aimed at facing this challenge. The techniques focus respectively on identifying lines of code, parsing any structured fragment, and recognizing the kind of "languages" used in development emails, so that we can apply ad-hoc analysis techniques to exploit their peculiarities.

Detected lines of source code in emails. In Chapter 7, we conducted a first work toward finding the structure in email content. Frequently development emails pertaining to a software system report parts of text written in other languages, especially source code snippets or stack traces. We devised lightweight techniques that, on the basis of simple text inspections, exploiting characteristics of source code text, can detect source code fragments in emails, fast and with a high accuracy. A practitioner can precisely classify thousands of emails, even at run-time. We also proposed novel methods for classifying lines that enclose source code. Using refined approaches, based on those used for the complete document classification, our methods achieve performance higher than the ones previously obtained through complex machine learning techniques. Moreover, almost all methods we developed can be configured with a threshold parameter that allows choosing the best trade-off between precision and recall, according to the user's needs. To assess our techniques, we created manual benchmark. Using our benchmark, we also conducted a statistical analysis of the email content and assessed that the vast majority of source code fragments are mentioned as lines separated from the natural language text.

Recovered the structured fragments in textual artifacts. In Chapter 8, we implemented two approaches based on island parsing [136] to extract structured fragments embedded in natural language written documents. Our approaches, iLander and PetitIsland, tested on development mailing lists and Stack Overflow posts, showed extremely accurate results in extracting the whole structured information embedded in natural language text. We showed how this structured information can be successfully used to conduct novel system analyses, such as reconstructing an entire alternative model of a system by simply mining its mailing list, and analyzing the trends in discussion in a stream of textual documents. Our PetitIsland framework has been devised to be not only accurate and efficient, but also flexible and extensible. We showed how we used it, for example, to create lightweight parsers for source code files, to support software analysis based on text mining.

Classified the lines of development emails. In Chapter 9, we conducted a work to finely classify the content of development emails in different categories: natural language, source code, stack traces, patches, and junk (*i.e.*, irrelevant information). Given the diversity of languages used in the example email, we realized that by considering the content of emails as a single bags of words, we would obtain a motley set of flattened terms without a clear context, and we would severely reduce the quality and the amount of available information. On the contrary, by automatically distinguishing the parts that form the content of an email, we provide better support for many tasks, such as better traceability link recovering, better content parsing, and improved artifact summarization. We devised Mucca, an approach based on a combination of parsing techniques (based on our PetitIsland framework) and machine-learning methods, to classify the contents of development emails in the chosen five categories. Our technique works at the line level, which–by inspecting hundreds of emails–we found to be the appropriate granularity for email content classification. We evaluated the effectiveness of our approach on a manually created benchmark created from the mailing lists of four OSS projects. Using mailing list cross validation, we found that our approach reacher very high results both in precision and recall.

11.1.4 Updated our knowledge on OSS mailing list communication

In Chapter 10, we analyzed the communication that takes place in OSS development mailing list, with the aim of updating our view on the content they archive. In fact, even though mailing lists have been considered—historically—the hub of project communication, during our analyses we sensed a change in the usage of development mailing lists across OSS systems. This called for an extensive investigation of the status of mailing list communication. We qualitatively analyzed the threads exchanged in the development mailing list of Lucene, across its entire history. We found that indeed the role of the development mailing list changed, by diminishing its importance in OSS project communication. At the same time, the issue repository (*i.e.*, Jira, in the case of Lucene) is emerging as a very interesting source of unstructured data. In fact, this repository is used no longer only for submitting requests for fixing defects, but also by users and developers to interact, set new milestones, and discuss about the details of the project. Given how we devised our approaches, the techniques we presented in this dissertation to deal with mailing list data can be adapted to issue repository data, which is less noisy and more structured than email messages.

11.1.5 Benchmarks

As an additional contribution, we implemented a number of benchmarks to support the research presented in this dissertation.

Linking emails and code. For the study in Chapter 4, we produced two benchmarks for evaluating the recovery of traceability links between emails and source code artifacts. We created them by analyzing the mailing lists of six diverse OSS systems written in four different programming languages. It includes more than 5,000 manually annotated emails, with links to all the code artifacts considering the whole history of the analyzed systems.

Recognizing source code lines. For the study in Chapter 7, we produced a benchmark that features sets of sample emails, randomly extracted from five unrelated JAVA OSS systems, which we manually read to label structured fragments, using our MILER GAME application. It includes more than 1,800 emails in which all the lines of code are manually labeled.

Recognizing structured fragments in emails. For validating ILANDER (Section 8.3) we produced a benchmark for evaluating the recognition, extraction, and modeling of JAVA content in email data. We created it by analyzing sample emails from four unrelated JAVA OSS systems, which we manually read to label and describe embedded structured fragments. It comprises 188 labeled emails with described structured fragments.

Recognizing source code fragments in Stack Overflow posts. For validating our second approach to island parsing, (*i.e.*, PETITISLAND, presented in Section 8.7), we adapted and improved a previously published benchmark to assess the recognition of source code fragments in Stack Overflow posts. It comprises 188 posts embedding more than 350 code fragments.

Classifying lines of development emails. To evaluate our MUCCA approach, we produced a benchmark to evaluate the classification of email lines into five categories, *i.e.*, natural language, source code, patch, stack trace, and noise (see Chapter 9). It features more than 1,400 emails comprising almost 69,000 manually classified lines.

Aliasing resolution. To conduct statistical analyses in our qualitative work on mailing list communication (see Chapter 10), we had to manually resolve aliases for participants. This benchmark comprises the resolution of the aliases for 200 email addresses, and can be used to evaluate automatic technique to solve the same task.

Development email categorization. In Chapter 10 we produced a manual classification of development mailing list threads into categories of discussions. We used card sorting to create the categories and we sorted more than 500 cards. This dataset can be used to evaluate techniques for automatic classification or for additional data analyses.

11.2 Future Work

In our thesis, we showed that natural language documents–if correctly mined, measured, and made available–can integrate, consolidate, and complement the data extracted from structured sources, because they include human factors and can be used as a source of qualitative data. The repositories that store artifacts with textual narrative, however, are still largely unexplored by researchers and not exploited by software developers. On the one hand practitioners do not employ textual artifacts during development for many reasons: Developers do not know whether a specific topic is expressed in these artifacts, fast full-text search is not always implemented, different kinds of artifacts provide different search and browsing tools, there is no consistency among different repositories and tools, and it is hard to know whether an artifact contains updated information or not. On the other hand, researchers must still find appropriate techniques for extracting relevant data by parsing natural language text. Our vision is that in the next years we will have to tackle both issues: We will have to make our results more accessible to practitioners (for example by devising recommender systems and relying on multiple source of unstructured data seamlessly), and we will continue to refine our ability to extract *relevant* information from unstructured content.

During the work on this dissertation, we also encountered promising future short-term research directions. Some of them are ideas on how to overcome limitations of our approach. In the following, we outline possible future work, discussing also—when appropriate—the shortcomings that originate them.

Improved Traceability Links. In our work on REmail, we verified its usefulness for program comprehension: By reading emails related to the classes at hand, we could improve our specific and general program knowledge. At the same time, this work allowed us to understand how some of the emails related to an entity have more significance than others. In emails, entities can be mentioned in code snippets, patches, stack traces, or in natural language sentences. We noticed that, in most of the cases, entities mentioned in natural language sentences give us true *qualitative* information: Stack traces can be a list of classes with no special meaning, but a class included in a broader natural language sentence is often well contextualized and explained. Thanks to our work on Mucca, we can answer quantitative questions about mailing list content and usage (*e.g.*, are developers sharing code? Or, do users report stack traces in mailing list?), and we can give a classification to our traceability links. Links would be associated with a tag distinguishing the text in which the entity is referenced, *e.g.*, natural language or stack trace. We believe that links with natural language tags might be useful for qualitative analysis, while links with other tags might be useful for quantitative analysis.

Chat coupling. Change coupling [76] detects evolutionary and implicit dependencies among artifacts of a system, by analyzing the history of their co-changes. We believe that we might analogously relate the artifacts that are often discussed together: If classes are mentioned in the same discourse, they could be logically connected. This information could reinforce the same structure we find in the source code, could confirm change coupling information, or could give insights on unexpected implicit dependencies. On top of the classification provided by Mucca, we plan to conduct a case study to test the importance of this metric for software engineering.

Opinion metrics. Our work on popularity, which relates the email popularity and defects of code artifacts, relies on simple quantitative data. Popularity counts the number of emails discussing about a class, the number of threads, the number of authors, *etc.*, but it does not seize *what* these authors in these emails say. We plan to improve our work by performing analysis of the *opinions* and *sentiments*, including methods from the IR and natural language processing fields (*e.g.*, [58; 207]). Our target is understanding whether email authors are sharing an opinion about a source code entity: We expect negative opinions to better correlate with defects than mere number of emails, or give us more information about where we can improve a software system. This work requires a survey of the state of the art in opinion mining and sentiment analysis, in order to find the most appropriate approach for our domain. From our first analysis, we realized that it could be conducted only with the classification provided by Mucca in place, as these techniques can only be applied to clean natural language text.

Events and trends in natural language artifacts. Our work on trend and event analysis is still very preliminary, for this reason we envision future work on this direction. The amount of natural language artifacts generated around a software system can form a very extensive amount of information. For example, in the Linux kernel mailing list, developers and users currently exchange more than 10,000 emails a month. Finding the most interesting or relevant information in this amount of data can be a daunting task, especially for not experts of the system. We believe we can adopt methods from the IR research field to help us to find the way to the most important details enclosed in NL artifacts that create temporal streams (*e.g.*, emails, issue reports, forum posts). More precisely, we are considering that the study of *unexpected events* and *emerging trends* might conduct us to interesting results. Our idea is that a moment in time in which we discover an unexpected event (*e.g.*, something almost never discussed appears) can be symptom of problems in the system or in the development team; similarly, an emerging trend might be a prelude of a change in the system. We believe that we can study these trends and events from a historical perspective to guide us to emails that could explain how and why a system reached a certain status. From a development point of view, we can keep track of emerging trends and unexpected events in projects we rely on. For example, if we use open source libraries, our analysis could monitor their mailing lists and issue repositories and alert us when we should be aware of upcoming changes, without the need of constantly reading the exchanged documents.

11.2.1 Lessons learned by mining development emails

To provide evidence toward our thesis, we decided to focus on development emails. Considering other data sources (*e.g.*, design documents) would have introduced a series of slight variations (*e.g.*, in our data importers and metamodels), thus potentially hindering our progress; we decided to solve the problem in the most appropriate way on a single data source.

Among the available repositories, we decided to analyze emails for a number of reasons: For example, because email text is completely free form, public software repositories provide a very large amount of email data pertaining to diverse software projects, and email communication has been central to the life of many industrial and OSS software systems.

During our research we have been witnessing a transition from mailing lists to more specialized communication channels. As also pointed out in Chapter 10, mailing lists are quickly losing out to issue trackers, code reviews, and online forums (such as Q&A websites). A practitioner, or a researcher, might be interested in applying the principles presented in this thesis to other data sources (*e.g.*, issue repositories, online forums, chat logs). To help porting the principles of our work to other sources and to fulfill the wider initial aim of the thesis, we present the lessons we have learned while mining development emails.

Know the data and the mining techniques. By mining development emails we found that techniques that worked correctly in previous research when applied other unstructured data sources (*e.g.*, for software requirements) cannot be applied *as they are* to mailing lists. For example, when conducting the research presented in Chapter 4, we used techniques from related work (*e.g.*, LSI and vector space model) to email data. We found that results were not satisfying as expected from reading previous applications. This was due to the fact that, although the used method was fully functional, we provided it with a very poor input data and suboptimal configuration settings. We came to this conclusion, only after a number of iterations and trial and errors, in which we often blamed the method to be inappropriate for the task.

The lesson learned is to not expect a method to work off-the-shelf on a novel data source. To achieve the best results, we first have to understand the theory on which the chosen method is based and how the data source we want to use as an input differs from data on which the method was previously used. For example, only when we acknowledged the relevance of noise in emails and the presence of named references to code artifacts, we could start tackling the problem with previously devised techniques in an effective way.

Everything needs to change, so everything can stay the same. The one thing we can count on in software development is *change*. We found that this holds also in explorative research on mining unstructured data. For this reason, we decided to implement Miler, our toolset, in a dynamically typed programming language, which allows faster modifications to the underlying metamodel, thus better supporting iterative explorative research. For the same reason, we found convenient to adopt a document-oriented database in our REmail implementation. On the other hand, however, we decided *not* to create single use throw away scripts to conduct our experiments, but we created a full-fledged toolset to explore, expose, and exploit email data. This greatly helped us to reuse components across experiments and build increasingly sophisticated solutions.

The lesson learned is that some software design principles can be applied to achieve successful research in mining unstructured data. In particular, one design principle suggests identifying the aspects of the application that vary and separate them from what stays the same; we found it essential to adapt this principle to our mining unstructured data research. Metamodels and approaches do vary, but they have to be integrated in a common flexible infrastructure, so that they can be taken to the next step.

Find the latent structure. In Part II we approached the linking between emails and source code artifacts considering email text as a mere bag of words. Even though we successfully employed a number of IR methods in conjunction with more lightweight textual matching techniques, we found it problematic to bring our research to the next step by only using this approach. In fact, bag of words do not consider the relation among words and, to give more meaningful results, statistical methods require a very large number of documents, which may not be available.

One of the most important lessons that we learned from this situation is that it is greatly beneficial to discover the latent structure of unstructured documents. Giving an appropriate structure to a textual document helps finding its real meaning and applying the most appropriate techniques for exposing the enclosed information. For example, as mentioned in the future work, knowing that a certain entity is mentioned in a certain block of text helps giving the traceability more or less relevance to the user. Giving a structure to unstructured documents is a very challenging endeavor and requires a deep knowledge of the studied documents, their content, their usage, and their peculiarities. We were able to achieve this only in a later time in our research. However, the practitioner or the young researcher should not be discouraged by this difficulty, because the achieved benefits will exceed the initial effort spent.

11.3 Closing Words

In this dissertation, we showed that exploiting the information gathered from unstructured data produced during the evolution of a software system leads to novel techniques for both software evolution analysis and for supporting program comprehension tasks. We showed that software evolution should not be considered as a process where the changing source code is the only important aspect: Software evolution is a complex process with many dimensions, which leave traces in distinct repositories. Unstructured data, written by people for the other people involved in the life of a software project, opens new perspectives on the evolution. However, the best results are achieved when the data from novel repositories is connected to the code.

This dissertation, while showing the importance of mining and integrating unstructured software data in our analyses and development, is only another step toward capturing software evolution as a complex and holistic phenomenon.

Bibliography

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *Proceedings of ICPC 2008 (16th IEEE International Conference on Program Comprehension)*, pages 103–112, 2008.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of CASCON 2008*, pages 304–318. ACM, 2008.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[4] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9(1-4):35–58, 2000.

[5] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*, pages 298–308, 2009.

[6] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley and Sons, 1988.

[7] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 95–104. ACM, 2010.

[8] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 324–333. ACM, 1999.

[9] A. Bacchelli, L. Baracchi, and M. Lanza. Remail -blending talk and work in eclipse. In *In Proceedings of Eclipse-IT 2011 (6th Workshop of the Italian Eclipse Community)*, pages 303–306, 2011.

[10] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci. Extracting structured data from natural language documents with island parsing. In *In Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, pages 476–479, 2011.

[11] A. Bacchelli, T. dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *In Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385, 2012.

[12] A. Bacchelli, M. D'Ambros, and M. Lanza. Are popular classes more defect prone? In *Proceedings of FASE 2010 (13th International Conference on Fundamental Approaches to Software Engineering)*, pages 59–73, 2010.

[13] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering)*, pages 205–214. IEEE CS Press, 2009.

[14] A. Bacchelli, M. Lanza, and M. D'Ambros. Miler - a tool infrastructure to analyze mailing lists. In *Proceedings of FAMOOSr 2009 (3rd International Workshop on FAMIX and Moose in Reengineering)*, 2009.

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

[16] A. Bacchelli, M. Lanza, and V. Humpa. Towards integrating e-mail communication in the IDE. In *Proceedings of SUITE 2010 (2nd International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation)*, pages 1–4, 2010.

[17] A. Bacchelli, M. Lanza, and V. Humpa. RTFM (Read The Factual Mails) –augmenting program comprehension with remail. In *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)*, pages 15–24, 2011.

[18] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 375–384. ACM Press, 2010.

[19] T. Ball, J.-M. K. Adam, A. P. Harvey, and P. Siy. If your version control system could talk. In *Workshop on Process Modelling and Empirical Studies of Software Engineering 1997 (19th International Conference on Software Engineering)*. IEEE Computer Society Press, 1997.

[20] I. Barker. What is information architecture? `http://www.steptwo.com.au/`, May 2005.

[21] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[22] H. Basten and P. Klint. Defacto: Language-parametric fact extraction from source code. In *Proceedings of SLE 2008 (International Conference of Software Language Engineering)*, pages 265–284. Springer, 2008.

[23] O. Baysal and A. J. Malton. Correlating social interactions to release history during software evolution. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, page 7. IEEE Computer Society, 2007.

[24] M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. Factorial hidden markov models. In *Machine Learning*, pages 29–245. MIT Press, 1997.

[25] A. L. Berger, V. J. Della Pietra, and S. A. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[26] T. Bergin and R. G. (Eds.). *History of Programming Languages-II*. Addison-Wesley, 1996.

[27] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Proceedings of IWPSE 2007 (9th International Workshop on Principles of Software Evolution (International Workshop on Principles of Software Evolution)*, pages 11–18. IEEE CS Press, 2007.

[28] M. Berry and M. Browne. *Understanding Search Engines - Mathematical Modeling and Text Retrieval*. SIAM, 2nd edition, 2005.

[29] M. W. Berry, S. T. Dumais, and T. A. Letsche. Computational methods for intelligent information access. In *Proceedings of SC 1995 (ACM/IEEE Conference on Supercomputing)*, 1995.

[30] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of SIGSOFT '08/FSE-16 (ACM SIGSOFT International Symposium on Foundations of software engineering)*, pages 308–318. ACM, 2008.

[31] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of MSR 2008 (5th IEEE Working Conference on Mining Software Repositories)*, pages 27–30. ACM, 2008.

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

[33] N. Bettenburg, S. W. Thomas, and A. E. Hassan. Using code search to link code fragments in discussions and source code. In *Proceedings of CSMR 2012 (16th European Conference on Software Maintenance and Reengineering)*, pages 319–329. IEEE, 2012.

[34] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. 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*, pages 121–130. ACM, 2009.

[35] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in OSS projects. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, pages 26–29. IEEE Computer Society, 2007.

[36] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of MSR 2006 (3th International Workshop on Mining Software Repositories)*, pages 137–143. ACM, 2006.

[37] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks in Postgres. In *Proceedings of MSR 2006*, pages 185–186, 2006.

[38] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Wngineering)*, pages 24–35. ACM, 2008.

[39] D. M. Blei, A. Y. Ng, M. I. Jordan, and J. Lafferty. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[40] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1):91–121, 1999.

[41] R. G. Burgess. *In the Field: An Introduction to Field Research.* Unwin Hyman, 1st edition, 1984.

[42] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of ICML (23rd International Conference on Machine learning)*, pages 161–168. ACM, 2006.

[43] V. R. Carvalho and W. W. Cohen. Learning to extract signature and reply lines from email. In *Proceedings of CEAS 2004 (1st Conference on Email and Anti-Spam)*, 2004.

[44] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of ESEM '08 (the Second ACM-IEEE international symposium on Empirical software engineering and measurement)*, pages 2–11. ACM, 2008.

[45] X. Chen and J. Grundy. Improving automated documentation to code traceability by combining retrieval techniques. In *Proceedings of ASE 2011 (26th IEEE/ACM International Conference on Automated Software Engineering*, pages 223–232, 2011.

[46] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[47] J. Conklin and M. L. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. In *Proceedings of CSCW 1988 (3rd ACM conference on Computer-supported cooperative work)*, pages 140–152. ACM, 1988.

[48] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[49] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM System Journal*, 28(2):294–306, 1989.

[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[51] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[52] M. D'Ambros. *On the Evolution of Source Code and Defects*. PhD thesis, University of Lugano, Switzerland, Oct. 2010.

[53] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Proceedings of QSIC 2010 (10th International Conference on Quality Software)*, pages 23–31. IEEE CS Press, 2010.

[54] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analyzing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. Springer, 2008.

[55] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pages 227–236. IEEE CS Press, 2006.

[56] M. D'Ambros, M. Lanza, and M. Pinzger. The metabase: Generating object persistency using meta descriptions. In *Proceedings of FAMOOSR 2007 (1st Workshop on FAMIX and Moose in Reengineering)*, 2007.

[57] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[58] K. Dave, S. Lawrence, and D. M. Pennock. Mining the peanut gallery: opinion extraction and semantic classification of product reviews. In *Proceedings of WWW 2003 (12th international conference on World Wide Web)*, pages 519–528. ACM, 2003.

[59] A. De Lucia, R. Oliveto, and G. Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *In Proceedings of ICSE 2008 (30th ACM/IEEE International Conference on Software Engineering)*, pages 839–842, 2008.

[60] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[61] A. Dekhtyar and J. Hayes. Good benchmarks are hard to find: Toward the benchmark for information retrieval applications in software engineering. In *ICSM 2006 Working Session: Information Retrieval Based Approaches in Software Evolution*, 2007.

[62] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proceedings of MSR 2004 (1st International Workshop on Mining Software Repositories)*, pages 22–26, 2004.

[63] S. Ducasse, L. Renggli, D. Shaffer, R. Zaccone, and M. Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[64] N. Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *CSCW*, 14(4):323–368, 2005.

[65] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[66] K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.

[67] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[68] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Ticky, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, 2005.

[69] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of ICSM 2003 (19th IEEE International Conference on Software Maintenance)*, pages 23–32. IEEE Computer Society, 2003.

[70] R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *Proceedings of ICSM 1998 (14th IEEE International Conference on Software Maintenance)*, pages 94–102. IEEE Computer Society, 1998.

[71] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[72] K. Fogel. *Open Source Development with CVS*. Cariolis Open Press, November 1999.

[73] K. Fogel. *Producing Open Source Software*. O'Reilly Media, first edition, 2005.

[74] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, Jan. 2004.

[75] R. Frost. Jazz and the eclipse way of collaboration. *IEEE Software*, 24(6):114–117, 2007.

[76] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of IWPSE 2003*, pages 13–. IEEE CS, 2003.

[77] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*. ACM Press, 2004.

[78] T. Gilb. *Software Metrics*. Winthrop, 1977.

[79] R. Gillespie. *Manufacturing Knowledge: A History of the Hawthorne Experiments*. Cambridge University Press, 1st edition, 1993.

[80] T. Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, November 2005.

[81] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies For Qualitative Research*. Aldine, 1967.

[82] T. Gleixner. The realtime preemption patch: Pragmatic ignorance or a chance to collaborate? In *Keynote of ECRTS 2010 (22nd Euromicro Conference on Real-Time Systems)*, 2010. `http://lwn.net/Articles/397422/`.

[83] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Oracle, 4th edition, 2012.

[84] T. L. Graves and A. Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Symposium on Software §Metrics (METRICS 1998)*, pages 276–273. IEEE Computer Society, 1998.

[85] C. Gutwin, R. Penner, and K. A. Schneider. Group awareness in distributed software development. In *Proc. of CSCW'04*, pages 72–81, 2004.

[86] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen. Communication in open source software development mailing lists. In *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)*, page to be published, 2013.

[87] A. Guzzi, A. Begel, J. K. Miller, and K. Nareddy. Facilitating enterprise software developer communication with cares. In *Proc. of ICSM'12*, pages 527–536, 2012.

[88] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[89] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 223–226. ACM, 2010.

[90] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[91] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[92] A. E. Hassan. The road ahead for Mining Software Repositories. In *Proceedings of FoSM 2008 (Frontiers of Software Maintenance)*, pages 48–57, 2008.

[93] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of ICSE 2009 (31st International Conference on Software Engineering)*, pages 78–88, 2009.

[94] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of ICSM 2005 (21st IEEE International Conference on Software Maintenance)*, pages 263–272. IEEE CS, 2005.

[95] A. E. Hassan, R. C. Holt, and A. Mockus. MSR 2004: International workshop on mining software repositories. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 770–771. IEEE Computer Society, 2004.

[96] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Notices*, 24(11):43–75, 1989.

[97] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of MSR 2008 (5th Working Conference on Mining Software Repositories)*, pages 23–26. ACM Press, 2008.

[98] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan. Should I contribute to this discussion? In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 181–190. IEEE CS Press, 2010.

[99] K. S. Jones. Automatic summarising: The state of the art. *Information Processing and Management*, 43:1449–1481, 2007.

[100] J. B. Jr., E. J. Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 177–186. ACM, 2005.

[101] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 2nd edition, 2009.

[102] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineeering)*, page to be published, 2011.

[103] M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proceedings of FSE 2006 (16th SIGSOFT Symposium on the Foundations of Software Engineering)*, pages 1–11. ACM Press, 2006.

[104] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings AOSD 2005 (4th international conference on Aspect-oriented software development)*, pages 159–168. ACM, 2005.

[105] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proceedings of ICSE 2007*, pages 489–498. ACM, 2007.

[106] V. C. Klema and A. J. Laub. The Singular Value Decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176, 1980.

[107] A. Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174. ACM Press, 2000.

[108] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[109] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 344–353. IEEE Computer Society, 2007.

[110] R. Kollmann, P. Selonen, and E. Stroulia. A study on the current state of the art in toolsupported uml-based static reverse engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 22–32, 2002.

[111] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[112] K. Kuwabara. A bazaar at the edge of chaos. *First Monday*, 5(3), 2000.

[113] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[114] T. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pages 492–501. ACM, 2006.

[115] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, (9):1060–1076, 1980.

[116] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.

[117] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineerng*, 10:311–341, 2005.

[118] P. L. Li, J. Herbsleb, and M. Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: A case study of openbsd. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 32. IEEE Computer Society, 2005.

[119] W. Lidwell, K. Holden, and J. Butler. *Universal Principles of Design*. Rockport, 2003.

[120] M. Lormans and A. van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 47–56, 2006.

[121] M. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, Switzerland, Oct. 2009.

[122] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[123] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[124] T. D. Marco. *Peopleware - Productive Projects and Teams*. Dorset House, 1999.

[125] A. Marcus, A. De Lucia, J. H. Hayes, and D. Poshyvanyk. Working session: Information retrieval based approaches in software evolution. In *Proceedings of ICSM 2006 (22th IEEE International Conference on Software Maintenance)*, pages 197–209. IEEE CS Press, 2006.

[126] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of ICSE 2003 (25th International Conference on Software Engineering)*, pages 125–135. IEEE CS Press, 2003.

[127] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[128] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM 2004 (20th IEEE International Conference on Software Maintenance)*, pages 350–359. IEEE Computer Society Press, 2004.

[129] B. Martin and B. Hanington. *Universal Methods of Design*. Rockport, 2012.

[130] R. McNaughton and H. Yamada. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[131] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[132] H. Mills. Software development. *IEEE Transactions on Software Engineering*, 2, 1976.

[133] T. Mitchell. *Machine Learning*. McGraw Hill, 1nd edition, 1997.

[134] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A case study of open source software development: the apache server. In *Proc. of ICSE'00*, pages 263–272, 2000.

[135] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[136] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[137] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, pages 181–190, 2008.

[138] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[139] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[140] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*, pages 580–586. ACM, 2005.

[141] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*, pages 284–292. ACM, 2005.

[142] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the ICSE 2006 (28th International Conference on Software Engineering)*, pages 452–461. ACM, 2006.

[143] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of ICSE '08 (30th international conference on Software engineering)*, pages 521–530. ACM, 2008.

[144] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of CCS 2007 (14th ACM Conference on Computer and Communications Security)*, pages 529–540. ACM, 2007.

[145] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005 (5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*, pages 1–10. ACM Press, 2005.

[146] M. Ogawa, K.-L. Ma, C. Bird, P. T. Devanbu, and A. Gourley. Visualizing social interaction in open source software projects. In *6th International Asia-Pacific Symposium on Visualization*, pages 25–32, 2007.

[147] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[148] P. W. Oman and T. G. Lewis. *Milestones in Software Evolution*. IEEE Computer Society Press, 1990.

[149] A. Oram and G. Wilson. *Making Software*. O'Reilly, 2010.

[150] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Software Engineering Notes*, 27(4):55–64, 2002.

[151] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proceedings of ISSTA 2004 (ACM SIGSOFT International Symposium on Software testing and analysis)*, pages 86–96. ACM, 2004.

[152] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[153] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Automating algorithms for the identification of fault-prone files. In *Proceedings of ISSTA 2007 (ACM SIGSOFT International Symposium on Software testing and analysis*, pages 219–227. ACM, 2007.

[154] D. Pattison, C. Bird, and P. Devanbu. Talk and Work: a Preliminary Report. In *Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)*, pages 113–116. ACM, 2008.

[155] A. Perer, B. Shneiderman, and D. W. Oard. Using rhythms of relationships to understand e-mail archives. *Journal of the American Society for Information Science and Technology*, 57:1936–1948, 2006.

[156] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

[157] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Proceedings of CSMR 2013 (17th European Conference on Software Maintenance and Reengineering)*, page to be published. IEEE CS Press, 2013.

[158] H. Ramankutty. Yacc/bison - parser generators - part 1. *Linux Gazette*, 87, 2003. `http://tldp.org/LDP/LGNET/issue87/ramankutty.html`.

[159] B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, 1992.

[160] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 505–514. ACM, 2010.

[161] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004*, page 223. IEEE CS, 2004.

[162] E. Raymond. *The Cathedral and the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.

[163] L. Renggli, S. Ducasse, Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *Proceedings of DYLA 2010 (4th Workshop on Dynamic Languages and Applications)*, 2010.

[164] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[165] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the Apache server. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, pages 541–550. ACM, 2008.

[166] P. C. Rigby and A. E. Hassan. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, pages 23–. IEEE Computer Society, 2007.

[167] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, pages 832–841. ACM, 2013.

[168] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, pages 541–550. ACM, 2011.

[169] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27:80–86, 2010.

[170] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[171] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of ICSE 2009 (31st International Conference on Software Engineering)*, pages 23–33. IEEE Computer Society Press, 2009.

[172] A. Schröter, J. Aranda, D. Damian, and I. Kwan. To talk or not to talk: factors that influence communication around changesets. In *Proc. of CSCW'12*, pages 1317–1326, 2012.

[173] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of MSR 2008*, pages 121–124. ACM, 2008.

[174] R. C. Seacord, D. Plakosh, , and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[175] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25:557–572, 1999.

[176] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34:1–47, 2002.

[177] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of ICSE 1996 (18th international conference on Software engineering)*, pages 387–396. IEEE Computer Society, 1996.

[178] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the use of developer irc meetings in open source projects. In *Proceedings of ICSM 2009 (25th IEEE International Conference on Software Maintenance)*, pages 147–156. IEEE CS Press, 2009.

[179] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of ICSE 2003 (25th International Conference on Software Engineering)*, pages 74–83. IEEE Computer Society, 2003.

[180] M. Smith, D. Weiss, P. Wilcox, and R. Dewar. The OPHELIA traceability layer. In *Cooperative Methods and Tools for Distributed Software Processes (2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes)*, pages 150–161. FrancoAngeli, 2003.

[181] Software Composition Group. The FAMIX official website, 2010. `http://www.moosetechnology.org/docs/famix/3.0`.

[182] O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proceedings of the 12th Conference on Computational Linguistics*, pages 636–641, 1988.

[183] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[184] Sun Microsystems, Inc. Code conventions for the Java™programming language, 1999. `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`.

[185] J. Tang, H. Li, Y. Cao, and Z. Tang. Email data cleaning. In *Proceedings of KDD 2005 (11th ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 489–498. ACM, 2005.

[186] R. Tang, A. E. Hassan, and Y. Zou. Techniques for identifying the country origin of mailing list participants. In *Proc. of WCRE 2009*, pages 36–40. IEEE CS Press, 2009.

[187] A. E. H. Thanh H. D. Nguyen, Bram Adams. A case study of bias in bug-fix datasets. In *Proceedings of WCRE 2010 (17th IEEE Working Conference on Reverse Engineering)*, pages 259 –268. IEEE CS Press, 2010.

[188] S. W. Thomas. *Mining Unstructured Software Repositories Using IR Models*. PhD thesis, School of Computing, Queen's University, Canada, December 2012.

[189] G. C. M. Thomas Fritz, Jingwen Ou and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 385–394. IEEE Computer Society, 2010.

[190] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[191] W. Tichy. Design, implementation, and evaluation of a Revision Control System. In *Proceedings of the 6th International Conference on Software Engineering (ICSE 1982)*, pages 58–67. IEEE Computer Society Press, 1982.

[192] M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*, IJCAI'85, pages 756–764, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[193] M. Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[194] D. Cubranĭć, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[195] M. van Den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM TOSEM*, 12(2):152–190, 2003.

[196] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of CC 01 (International Conference on Compiler Construction)*, pages 365–370. Springer, 2001.

[197] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 365–370, London, UK, UK, 2001. Springer-Verlag.

[198] G. van Rossum. Unified diff format, June 2006. `http://www.artima.com/weblogs/viewpost.jsp?thread=164293`.

[199] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of ICSE 2003*, pages 408–418, 2003.

[200] G. Venolia. Textual allusions to artifacts in software-related repositories. In *Proceedings of MSR 2006*, pages 151–154. ACM, 2006.

[201] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineeering)*, pages 551 – 560. ACM Press, 2011.

[202] I. H. Witten, G. W. Paynter, E. Frank, and C. G. C. G. Nevill-Manning. KEA: practical automatic keyphrase extraction. In *Proceedings of DL 1999 (4th ACM conference on Digital Libraries)*, pages 254–255. ACM, 1999.

[203] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of ICSE 2009 (31th International Conference on Software Engineering)*, pages 1–11. IEEE Computer Society, 2009.

[204] T. Xie and J. Pei. MAPO: mining api usages from open source repositories. In *Proceedings of MSR 2006 (3th International Workshop on Mining Software Repositories)*, pages 54–57. ACM, 2006.

[205] K. C. B. Yakemovic and E. J. Conklin. Report on a development project use of an issue-based information system. In *Proceedings of CSCW 1990 (5th ACM conference on Computer-supported cooperative work)*, pages 105–118. ACM, 1990.

[206] S. S. Yau, J. S. Colofello, and T. MacGregor. Ripple effect analysis of software maintenance. In I. C. S. Press, editor, *Proceedings of the 2nd IEEE International Conference on Computer Software and Applications (COMPSAC 1978)*, 1978.

[207] J. Yi, T. Nasukawa, R. Bunescu, and W. Niblack. Sentiment analyzer: Extracting sentiments about a given topic using natural language processing techniques. In *Proceedings of ICDM 2003 (3rd IEEE International Conference on Data Mining)*, pages 427–. IEEE, 2003.

[208] A. Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software Engineering Notes*, 24(6):253–267, 1999.

[209] W. Zhao, L. Zhang, L. Yin, L. Jing, and J. Sun. Understanding how the requirements are implemented in source code. In *Proceedings of APSEC 2003 (10th Asia-Pacific Software Engineering Conference)*, pages 68–77. IEEE Computer Society, 2003.

[210] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, 2008.

[211] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering (TSE)*, 36(5):618–643, 2010.

[212] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of PROMISE 2007*, page 76. IEEE CS, 2007.

[213] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.