Supporting Newcomers in Software Development Projects

Doctoral dissertation at the
Department of Engineering,
University of Sannio, Benevento

presented by
Sebastiano Panichella

Under the supervision of:

Prof. Massimiliano Di Penta

and Prof. Gerardo Canfora

PhD Coordinator: Prof. Luigi Glielmo

July 2014

Acknowledgments

Summarizing more than three years of work and experience is not a trivial task and finding the right words to express acknowledgements is quite complex. I hope that they do not look like a set of ritual sentences, while I really feel and mean every single word. In such moments it is always difficult to find the words to thanks the people that make, in various and different ways, your life *more complete* and *more interesting*.

I think that the life require a professional realization. Working with people that you like it is a key and important aspect for that. Here at the Università degli Studi del Sannio, I met the first time the Prof. Massimiliano Di Penta and the Prof. Gerardo Canfora, my current advisors. They made me feel at home and work (work very hard) it has been easier because of this climate of familiarity. I feel different after these past three years with them. I owe my deepest gratitude to both of them during my research work. Thanks Max for everything (feedbacks and everything else)! Gerardo is professionally flawless and often his temperament reminds me my father. Thanks for all the advice during these three years! Special thanks to my examiners, Prof. Giuseppe Di Lucca (Università degli Studi del Sannio), Prof.ssa Elisabetta Di Nitto (Politecnico di Milano) and Prof. Andrian Marcus (Wayne State University) for managing to read this dissertation and for their helpful suggestions.

I can not forget that if I'm here I have to thanks the Prof. Andrea De Lucia and the Dr. Rocco Oliveto, I started to work for research with them (I believe in the best way). Rocco was the supervisor of my bachelor thesis and Andrea together with Rocco co-advised my master thesis. I thank them both. My sincere thanks also goes to Prof. Giuliano Antoniol at the École Polytechnique de Montrèal (Canada) for offering me the summer internship opportunities in his research group and for supporting me on diverse exciting works. During that internship I met Venera Arnaoudova, a very good friend, a very good worker and a promising researcher. I would like to thank Dr. Gabriele Bavota that currently works with me at the University of Sannio for his professionalism and strong propensity to research. Finally, I would like to

offer my thanks to all the people at the University of Sannio that made my PhD experience unforgettable.

I had the opportunities to mentor two bachelor students, Carmine Vassallo (University of Sannio) and Stefano Giannantonio (Università degli Studi del Molise). I would like to thank them for the high dedication and the excellent work performed in their bachelor thesis. I would like to thank also all the masters and bachelor students involved in the studies that we performed. I want to like to thanks also Jairo Aponte that worked with me in a work described in this thesis.

The first person that I remember of my family is my father, that taught me how is important to build true relationships and avoid people that do not believe in the importance of the human relationships. He was right, as always: true relationships makes the life of everyone special! He is the person who has always believed in me, I will never forget him, because he is the person who most influenced my life and helped me to find my way. I miss him so much! In a similar way, my mother encouraged me during all the steps of my life, she is able to teach me how is important to be brave and take important decisions in very critical situations. Her sweetness and her modernity together represent something that can be reached only after a long inner maturation. When I talk with her I think very often that is very difficult to find people with her honesty, her integrity and her dynamism. I bring with me the principles that she gave me always and everywhere, because now these are part of me. Thanks Mum! My brother Annibale is everything for me: my best friend, a good colleague and in my opinion, a genius in math. Because of him, I can claim that having a twin is one of the gifts that God has given me. It is completely true that twins have a telepathy when they talk, we don't need a lot of words to understand what we want to tell each other. It is like talking with yourself without talk alone (who doesn't have a twin is not able to understand completely this). Special thanks to him for the support, admirable enthusiasms, honesty and help he always provide me. He was my source of strength when I was weak and vulnerable. The people who really love you are near you in times of trouble and are able to feel happiness for you when something good happens in your life. Only few of the people in our life are like this. He is one of these. Thanks Annibale! My sister is a very dynamic person, I admire her ability to come out unscathed from any situation. She is a very sweet woman and lives to help the others. Temperamentally we are very similar and we do not like to take ourselves too seriously. We like to take more seriously the work and take care of the responsibilities that we have. This makes us remarkably similar even in the love life. Thanks Lucia for the wonderful moments together! About my private life, I was convinced that perfect love rela-

tionships don’t exist and moreover, to be honest I was convinced that no one girl would be right for me. Until I met you Cristiana. I dreamed for a long time of finding true love, the one that changes everything, what makes so wonderful the life. My life has a meaning now and it is only thanks to you. You’re my reason for living. I love you! Thanks for existing Cristiana!

Resume

Context: The recent and fast expansion of OSS communities has fostered research on how open source projects evolve and how their communities interact. Specifically, several research studies highlight how in OSS projects the inflow of new developers plays an important role in the longevity and the success of open-source software. Indeed, the longevity of such projects depend on the successful turnover between new and old developers, because without replacing members who leave, a community will eventually wither away. However, such research effort did not generate yet concrete results in support retention and training of project newcomers.

Aim: The objective of this thesis is (i) to study how newcomers behave during program comprehension activities and how they interact with others developers then, (ii) to develop tools for supporting them during development activities and in the integration in the development team. Hence, the thesis is composed by two main parts. Specifically, a first part of the thesis is aimed at understanding what kind of information can be obtained by analyzing data from software repositories (e.g., versioning systems) to help newcomers to collaborate with others developers and support the team work. The second part of the thesis, on the basis of insights obtained in the previous part, presents recommenders to concretely support project newcomers.

Method: we analyze unstructured developers' discussions—e.g., recorded in mailing lists, issue trackers, or chat—to extract the social links between developers, and we look at their code changes in the versioning system. Our purpose is to investigate how collaboration links (and recommendations generated from these links) vary and complement each other when they are identified through different communication channels. Moreover, we analyze navigation patterns among software documents to help newcomers to navigate software artifacts properly, and how developers summarize software artifacts through keywords. Finally, to support newcomers in their activities, we propose recommenders to (i) suggest mentors and (ii) to mine source code descriptions from discussion forums.

Results: The analysis of software repositories reveals the usefulness of mailing lists and

issue trackers in identifying developers coordinators in OSS projects. Furthermore, when analyzing how developers browse software artifacts during maintenance tasks we discovered that while more experienced developers follow an "integrated" approach while browsing software artifacts, junior developers might not exploit all the available documentation. We also found that the code summaries developers produced reflect high-frequency words or words part of topics having a high frequency. When evaluating the mentor recommender, we found that it is able to achieve a precision of about 75% or above, and that mailing lists and discussion forums represent a precious source of information for re-documenting source code and aid program comprehension.

Conclusions: In this thesis we investigated problems arising when newcomers join software projects, and possible solutions to support them. After a deep analysis of software repositories we found that it is possible to support the newcomer training with various recommenders. Among the many contributions of this thesis, we support the first newcomers training stage with the suggestion of appropriate mentors; then, we help newcomers during maintenance activities improving their program comprehension with the generation of high quality source code summaries or identifying descriptions in natural language (mined from developers' discussion) describing source code elements. For future work, we plan to improve the proposed recommenders and to integrate other kind of recommenders that can improve the newcomers training.

List of Publications

List of the Journal Publications of the candidate.

[1] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A, Panichella, S. Panichella. **Defect Prediction as a Multi-Objective Optimization Problem**. Actually considered with Major revision in Software Testing, Verification and Reliability (STVR) 2014 *doi:10.1109/ICST.2013.38*.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella. **How the Apache Community Upgrades Dependencies: An Evolutionary Study**. Submitted to Empirical Software Engineering (EMSE) 2014. *doi:10.1007/s10664-014-9325-9*.

[3] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella. **Labeling Source Code with Information Retrieval Methods: An Empirical Study**. *Empirical Software Engineering (EMSE)*, pp. 1−38, 2013. *doi:10.1007/s10664-013-9285-5*

[4] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, S. Panichella. **Improving IR-based traceability recovery via noun-based indexing of software artifacts**. *Journal of Software: Evolution and Process (JSE)*, vol. 25, no. 7, pp. 743−762, 2012. *doi:10.1002/smr.1564*

[5] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella. **Applying a Smoothing Filter to Improve IR-based Traceability Recovery Processes: An Empirical Investigation**. *Information and Software Technology (INFSOF)*, vol. 55, no. 4, pp. 741−754, 2012. *doi:10.1016/j.infsof.2012.08.002*

List of the Conference Publications of the candidate (full papers).

[6] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, G. Antoniol. **How Developers' Collaborations Identified from Different Sources Tell us About Code Changes.**. In: *Proceedings of the 30th International Conference on Software Maintenance and Evolution* (ICSME 2014). Victoria, Canada.

[7] S. Panichella, G. Canfora, M. Di Penta, R. Oliveto. **How the Evolution of Emerging Collaborations Relates to Code Changes: an Empirical Study**. In: *Proceedings of the 36th International Conference on Program Comprehension* (ICPC 2014), pp. 177−188. Hyderabad, India.

[8] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella. **An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks**. In: *Proceedings of the 29th International Conference on Software Maintenance* (ICSM 2013), pp. 210−219. Eindhoven, Netherlands.

[9] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella. **The Evolution of Project Inter-Dependencies in a Software Ecosystem: the Case of Apache.**. In: *Proceedings of the 29th International Conference on Software Maintenance* (ICSM 2013), pp. 80−89. Eindhoven, Netherlands.

[10] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella. **Multi-Objective Cross-Project Defect Prediction**. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation* (ICST 2013), pp. 52−61. Luxembourg.

[11] G. Canfora, M. Di Penta, R. Oliveto, S. Panichella. **Who is going to Mentor Newcomers in Open Source Projects?**. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2012), pp. 44:1−44:11. Cary, North Carolina, USA.

[12] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella. **Using IR Methods for Labeling Source Code Artifacts: Is It Worthwhile?**. In: *Proceedings of the 20th IEEE International Conference on Program Comprehension* (ICPC 2012), pp. 93−102. Passau, Germany.

[13] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, G. Canfora. **Mining source code descriptions from developer communications**. In: *Proceedings of the 20th IEEE International Conference on Program Comprehension* (ICPC 2012), pp. 63−72. Passau, Germany.

[14] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella. **Improving IR-based Traceability Recovery Using Smoothing Filters**. In: *Proceedings of the 19th IEEE International Conference on Program Comprehension* (ICPC 2011), pp. 21−30. Kingston, ON, Canada. **BEST PAPER AWARD**.

Conference Publication of the candidate (short paper).

[15] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, G. Canfora. **Recommending Refactorings based on Team Co-Maintenance Patterns.**. In: *29th international conference on Automated Software Engineering* (ASE 2014). Vasteras, Sweden.

List of the Conference Publications of the candidate (tool demo papers).

[16] C. Vassallo, S. Panichella, G. Canfora, M. Di Penta. **CODES: mining sourCe cOde Descriptions from developErs diScussions**. In: *Proceedings of the 36th International Conference on Program Comprehension* (ICPC 2014), pp. 106−109. Hyderabad, India. **BEST TOOL AWARD**.

[17] G. Canfora, M. Di Penta, S. Giannantonio, R. Oliveto, S. Panichella. **YODA: Young and newcOmer Developer Assistant**. In: *Proceedings of the 35th International Conference on Software Engineering* (ICSE 2013), pp. 1331−1334. San Francisco, CA, USA.

In the following, for each chapter it is specified the used materials (or part) of the publications presented in this thesis:

Chapter 2 Publication [6]

Chapter 3 Publication [7]

Chapter 4 Publications [2, 9]

Chapter 5 Publication [8]

Chapter 6 Publications [3, 12]

Chapter 7 Publications [11, 17]

Chapter 8 Publications [13, 16]

Contents

Chapter 1

Introduction

Open-source software (OSS) projects consist of very complex communities, where the participants, who are mostly volunteers, are distributed across different geographic locations. Developers of such OSS communities develop software in a public and collaborative manner relying on versioning systems. In addition, to make the communication between participants of a project easier, developers use mailing lists, IRC, and instant messaging, that are widely used as means of Internet communication between developers. With the main purpose to bug reporting help, and preserve stability of the software, developers use issue trackers such as Bugzilla[1] and Jira[2]. Thus, in OSS projects the software is developed, tested, or improved through public collaboration and distributed with the idea that the output of this process must be shared with others. Indeed, information about development discussions—as well as changes of source code—are publicity available as open source resources. However, a successful OSS project needs an appropriate and formal governance that supports the social-technical environment and the developers collaboration. For such reasons, development in OSS projects is often performed relying on services provided by software foundations that establish governance and structure of teams (in this way developers can align their work contributions with the project). Specifically, OSS foundations establish, first of all, a more effective way to structure projects; after that, it establish the quality criteria that the software must to comply to ensure the software usability, maintainability, portability and efficiency. In addition, a OSS foundation ensures the software quality by promoting the meritocracy and merit of developers who are active contributors to OSS projects. The base idea is that the community membership is granted only to those who actively/positively participate in a

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

[2] https://www.atlassian.com/software/jira

software project(s) in a collaborative manner.

Popular and successful examples of OSS foundations are the *Apache Software Foundation*[3](ASF), the *Eclipse Foundation* [4], the *Linux Foundation*[5] and the *Mozilla Foundation*[6]. In OSS projects, the inflow of new developers plays an important role in the longevity and the success of open-source software [18–20]. Thus, the longevity of such projects depend on the successful turnover between new and old developers. This means that a new generations of developers is crucial for the survival of a project over the time, because without replacing members who leave, a community will eventually wither away [21].

Kraut *et al.* [21] investigate the challenges of dealing with newcomers and point out several basic problems that online communities must solve:

- *Recruitment*: ensure a continuous influx of new developers;
- *Selection*: OSS communities try to select the newcomers that are more motivated (for example, the ASF newcomers support page point out that *"The more you give the more you get out"*) and who fit well the project needed [22–24];
- *Retention*: an OSS project needs to reduce the percentage of newcomers that leave the project because of the socio-technical barriers that they meet when they join the project;
- *Socialization*: "teaching" to newcomers how to behave with developers teams.

To ensure the growing of the community an OSS project must to *recruit* new developers to support the continuous turnover between new and old developers that are leaving the project. Several researchers investigated, the reasons behind the newcomers decision to abandon the project. Indeed, in general, a newcomer joins a software project encounters difficulties and obstacles when starting her contributions, resulting in a low permanence rate [19, 22, 25]. For example, previous studies show that there is a consequent low permanence rate of junior developers when they did not receive any answers (any support) from senior developers in the project (less than 20% of newcomers became long-term contributors) [19, 22]. The ASF to ensure the supply of newcomers has a newcomer support page[7] where are reported (i) a guide for first engagement with an Apache project (i.e., where a newcomer needs to start) and (ii) some common newcomer-related FAQ.

[3] http://www.apache.org/
[4] www.eclipse.org/
[5] www.linuxfoundation.org/
[6] www.mozilla.org
[7] https://community.apache.org/newcomers/

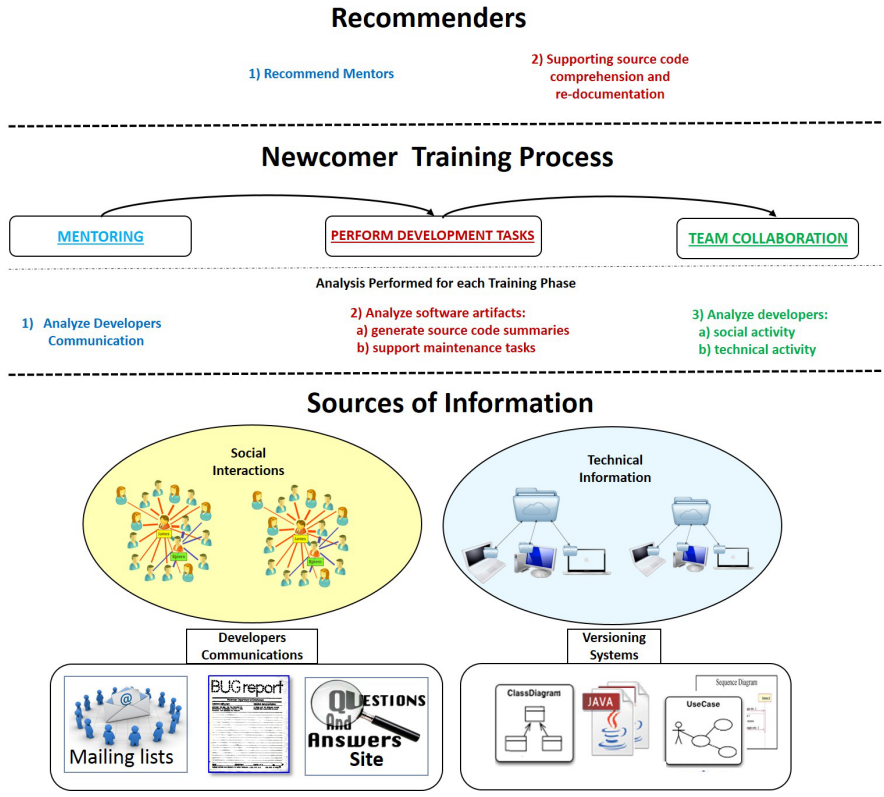


Figure 1.1: Newcomer Training Process: three high level phases.

When a newcomer is recruited, she needs a proper training to develop the skills and experience needed to work actively and autonomously in the project. There are various aspects that characterize a proper newcomers training: (i) project environment, (ii) newcomers expertise and known technologies, (iii) source code and documentation quality. Hence, some of them depend closely on the characteristics of the new developer, such as the "motivation" and the proneness to have bright ideas, while others aspects are, intrinsically, part of the newcomer training process [22]. Thus, it is possible try to support/improve the training process to help newcomers that are joining the project.

Figure 1.1 depicts the phases that characterize the newcomer training process [20, 22]: after a (i) *mentoring activity*, the newcomer (ii) *perform development tasks* benefited from the *team collaboration*. Specifically, the *mentoring activity* by experienced developers represents the initial part of the training process in which a newcomer is helped by mentors in OSS

project (in a formal Mentoring Programs and education) to acquire the technical and organizational information relevant for the project. Once the newcomer gained knowledge about the project organization, she needs to perform *maintenance/development tasks*. Clearly, gaining familiarity with the *social environment* to work properly with more experienced developers can help the newcomer to perform in the right way development activities. All of these phases are very interlinked and are important stages for a complete newcomer recruitment.

Past studies suggest that newcomers ties to the OSS communities are pretty fragile [18–22]. As a result, a community needs to help newcomers to become part of the community thus, have more robust ties to the community or learn how the group operates. In such context, Dagenais *et al.* [25] studied, by surveying 18 IBM developers, what happens when someone moves into a new "project landscape", making for her necessary to get acquainted with the new environment. Among the factors they found important, it is worthwhile to mention the need for early practicing, the availability of feedback for their work, and the need for getting initial guidance. Such studies confirm the importance of a **mentoring activity** for newcomers [19, 22, 25] to avoid problems/barriers that affect the newcomers participation. However, nothing concrete was proposed to help newcomers in this first stage of the newcomer training (Figure 1.1). For these reasons, this thesis investigates, as first contribution, the possibility to build a recommender able to suggest appropriate mentors to train project newcomers (Section 1.1 describes more in detail our contribution in this newcomer training phase).

Once newcomers are trained by appropriate mentors, they would gradually starts to commit changes in the source code repository. Thus, they want to gain familiarity with source code and the related documentation with the purpose to apply development/maintenance tasks (second phase in Figure 1.1). In such context program comprehension is preliminary to each development/maintenance activity. However, program comprehension is a very effort consuming task (especially in large projects), because very often, the source code lacks of comments that adequately describe its responsibility. Thus, this thesis, investigates the possibilities to **build high quality summaries of source code elements** to help new developers in program comprehension (Section 1.1 describes more in detail this contribution of the thesis). However, very often, source code documentation is scarce, incomplete, or out of date and this means that source code summaries (alone) are not sufficient to understand source code elements behavioral and responsibilities. We conjecture that in mailing lists and issue trackers discussions, developers discuss about code elements and explain their behavior and responsibilities. For this reason, in this thesis we also try to improve program comprehension and existing documentation by **mining source code descriptions from developer's communication** (Section 1.1 describe more in detail this contribution of the thesis).

Last but not the least, developers teams needs to make the newcomer familiar with the

environment and the project organization and "teaching" him/her how to behave with developers teams. However, dynamics are different in open source projects [26], that involve developers spread across the world and working in different time zones, often communicating using electronic means of communication, such as mailing lists. In essence, developers participating in open source projects are not staffed into teams by project managers. Moreover, the way they collaborate depends on the structure of the open source project. In such context, for a newcomer that arrived in a software project, gaining information about these socio-technical dynamics can play an important role in his/her permanence. Thus, OSS projects should carefully, help newcomers in understanding the dynamics of the social environment and the developer teams interaction to create the conditions where newcomers will be able to produce innovation in the team they enter. Traditional communication channels, like issue trackers and mailing lists are useful sources to extract relevant facts from the "social environment" of the project. For example, Developers' Social Network (DSN) mined from mailing lists can be analyzed to compute metrics to suggest the more active and more appropriate developers for the newcomers recuitment [27] while IRC chat can be useful to have on-line meeting and organize the work [28]. For such reason, in this thesis we focus our attention on the analysis of developers discussions in form of mailing lists, issue trackers and IRC chat with the aim to understand what kind of information can be extracted to support team work. Specifically, we **investigate the evolution of DSN** and its impact on software structure.
In summary, when a newcomer joins a project, she has to be trained from many different points of view: project architecture and implementation details, development guidelines, and organizational aspects. This first important stage of a newcomer is called *training phase*. This first stage is very important because once a newcomer has been trained, she can continue to work autonomously in the project. A newcomer would likely first start participating to discussion actively and then would gradually starts to commit changes in the source code repository. For these reasons, this thesis investigates and proposes various solutions/suggestions to help newcomers in both training and implementation phases. Such contributions are described more in detail in the Section 1.1.

1.1 Major Contributions

This section presents the research contributions of this thesis derived from the analysis of the research problems briefly described in the previous section. Specifically, we try to support newcomers during their recruitment analyzing developers communications in form of mailing lists, issue trackers and IRC chat with the aim to understand what kind of information can be extracted to support team work; (ii) investigating how developers browse and use soft-

ware artifacts to improve program comprehension; (iii) suggesting appropriate mentors to help newcomers during the training. As depicted in Figure 1.1, a complete newcomer training starts with a **mentoring support** by senior developers. Once, the newcomer is trained by a mentor(s) she continues with early **maintenance/development tasks**. While the newcomer applies development activities, they also improve their familiarity about the "social environment" to gain advantages from the **team collaboration**.

1.1.1 Analysis of Developers' Social Network to better understand developers interactions and Team Work

This section discusses the research contributions of this thesis derived from the analysis of DSN mined on different sources of information such as mailing lists, issue trackers, IRC chat, and versioning systems. The conjecture is that a newcomer by an accurate analysis of such communication channels can gain very important information about the "social project environment" helpful to support team work.

Once a newcomer has been trained by a proper mentor(s), she needs to gain information about the "social environment" to continue to work easily and autonomously in the project. Part of the knowledge can be gained asking information to mentors of the project. However, information gained from such mentors is sometimes insufficient because mentors in OSS project have very specific competencies. Hence, a newcomer is interest to ask other people specific questions that the initial mentor(s) is not able to answer. In other words, the newcomer needs to socialize with developers teams, "understand" how to behave in the right ways with developers teams and gain more information. The newcomer has to use communication channels, such as, issue trackers, mailing lists and IRC chat, to exchange information. Thus, DSN mined from mailing lists can be useful to identify coordinators in OSS projects [27]. From the other side, IRC chat can be useful to have on-line meeting and organize the work [28]. Vice versa, Twitter can be used to give general overview about the status of the project and the new and emerging technologies used in the project [29, 30]. However, developers of a OSS community tend to use some communication channels more than others. Also, collaborations identified through co-changes may differ from those identified through communication channels [31]. We argue that different sources can tell different/complementary information about developers' communication network.

This thesis provides an investigation about the usefulness of different recommendations based on different communication channels, such as mailing lists, issue trackers, or IRC chat. Specifically, the obtained results suggest the usefulness of some sources to identify coordinators and mentors in software projects, while other sources of information have a very high level of informational noise to be precise and concise enough in such recommendations.

Then, relying on such sources, we build DSN with the purpose of analyzing the evolution/reorganization of teams over the time. We discovered that changes in team structure are reflected in cohesive changes occurring in the source codes. This result not only explains the dynamics of the socio-technical congruence in software projects. Last, but not least, we analyze cross-project developers' communication networks and find how such communication is used to support upgrades of Third-Party Libraries(TPL).

1.1.2 Investigate How Developers Browse and Understand Software Artifacts to Improve Program Comprehension

Once a newcomer has been trained by appropriate mentor(s) and gained enough confidence with the "social environment", she is ready to become active from a technical point view, i.e., commit changes in the code to improve existing features and apply maintenance activity. Thus, a developer that just joined a project needs to acquire familiarity with the software system. We have investigated how developers use different sources of information during a program comprehension task, and, specifically (i) to what extent newcomers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts. Results indicate that, although newcomers spent an high proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Applying a more deeper analysis that considers the years of experience of the participants, we discover that more experienced newcomers follow an "integrated" approach by using different kinds of artifacts. Such information can be seen as a starting point to build recommenders that help the newcomer to properly navigate the software documentation when apply maintenance tasks.

We have also investigated how developers "summarize" source code elements through representative keywords, and carried out an empirical study aimed at investigating to what extent a source code labeling based on IR techniques would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task. Results of such a study show to what extent summaries produced by developers reflect high-frequency words (or topics) in the source code, and what kinds of code elements are being used to produce such summaries.

The lack of adequate source code documentation, developers try to infer this knowledge from external artifacts. We argue that messages exchanged among contributors/developers, in the form of issue trackers and emails, are a useful source of information to help understanding source code. However, such communications are unstructured and usually not explicitly meant to describe specific parts of the source code. Developers searching for code descrip-

tions within communications face the challenge of filtering large amount of data to extract what pieces of information are important to them. For these reasons, this thesis proposes an approach to automatically extract method descriptions of Java systems from discussions in mailing lists and issue trackers discussions. The extracted descriptions represent a useful source for program comprehension tasks. In addition, these descriptions can be seen as good candidate descriptions for code re-documentation.

1.1.3 Suggest Appropriate Mentors to Training Project Newcomers

When a newcomer joins a project, she has to be trained from many different points of view, such as project architecture and implementation details, development guidelines, and organizational aspects. Inadequate training could likely lead to project delay or failure. Thus, a proper newcomer training and a good project environment are highly desirable because can impact the probability of a newcomer to become a long term contributors [19, 22, 25]. A previous study by Steinmacher *et al.* [20] identified 38 barriers that affects the newcomer participation, grouped into seven different categories. Among the factors that influence negatively the permanence of a newcomer in OSS projects, the more important one is represented by the inadequate answers to their request to help. The Analysis of mailing list discussions highlighted that newcomers receive more answers by less experienced developers with respect to more experienced one. Moreover, Dagenais *et al* [25] surveyed software projects indicating that mentoring of project newcomers during their training phase is highly desirable [25]. For example, the Apache Software Foundation ensures a proper newcomer training applying a formal training program called *Mentoring Program*. Specifically, the page of the mentoring program[8] explains formally what a mentor has to do in that company and what a mentee can expect from her mentor. In that page the Apache Software Foundation asks committers to cover the role of Mentors for project newcomers. Thus, not all committers are mentors and senior developers have the free choice to become mentors or not of new developers. We argue that developers in OSS have some specific characteristics that make them mentors. Some of them are social/communication skills, technical competencies and knowledge about the project directives of the company. We believe that mailing lists, issue trackers and versioning systems are useful sources of information to identify mentors having these socio-technical skills. Steinmacher *et al.* [32] in according to our conjecture, claim that it is possible to recommend mentors is OSS projects relying on data from software repositories. This thesis provides an explicit and formal definition of "Mentor" in OSS projects and (stemming from the considerations of previous studies) presents an approach, named YODA (Young and newcOmer Developer Assistant) aimed at identifying and recommending mentors in software

[8] https://community.apache.org/mentoringprogramme.html

projects by mining data from mailing lists, issue trackers and versioning systems. Candidate mentors are identified among experienced developers who actively interact with newcomers. Then, when a newcomer joins the project, YODA recommends her a mentor that among the available ones, has already discussed topics relevant for the newcomer. Results show the potential usefulness of YODA as a recommendation system to aid project managers in supporting newcomers joining a software project.

1.2 Thesis Organization

This thesis is composed of three parts:

- **Part I**: in this part of the thesis we are interested to understand what kind of information can be derived by analyzing data from versioning systems and development discussions to help newcomers to collaborate with developers and support the team work. For this reason we analyze unstructured discussions between software developers, in form of mailing lists, issue trackers or IRC chat to extract the social interactions between project developers. We also analyze versioning systems to extract important facts about the code changes applied by experienced developers. Specifically, this part presents three studies, aimed at (i) identifying the more reliable communication channels to communicate with more experienced developers that cover important project roles (e.g., project coordinators); and (ii) investigating the evolution of DSN and its impact on software structure. Chapter 2 investigates the usefulness of different recommendations based on different communication channels. Specifically, we report a study that analyzes DSN and investigates how collaboration links vary and complement each other when they are identified through data from (i) three different kinds of communication channels, i.e., mailing lists, issue trackers, and IRC chat logs, and (ii) changes people performed on the same artifacts within close time frames. Chapter 3 observes how developers contributing to open source projects spontaneously group into "emerging" teams, reflected by messages exchanged over mailing lists and issue trackers. Moreover, the chapter investigates how, when a project evolves, emerging teams re-organize themselves (e.g., by splitting or merging). Chapter 4 studies the evolution of the Apache Software Ecosystem, in terms of number of developers, their interactions and the dependencies between projects and investigates (i) how dependencies between projects and the developers social links evolve over time when the ecosystem grows; (ii) how developers discuss the needs and risks of such upgrades.

- **Part II**: with the main goal to help newcomers in program comprehension task, this

part of the thesis investigates the information that can be derived analyzing source code and the interactions between software artifacts. For this reasons we analyze navigation patterns among use cases, class diagram, sequence diagrams, test cases to help newcomers to navigate properly software artifacts; then, using Information Retrieval (IR) methods we generate source code summaries to improve source code comprehension. Chapter 5 is aimed at investigating (i) to what extent newcomers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts. Chapter 6 motivates and reports an empirical study aimed at extract useful code summaries of source code artifacts with the aim of facilitating newcomers program comprehension.

- **Part III**: In this last part of the thesis, we (i) help newcomers during the training phase suggesting them proper Mentors; (ii) help project newcomers program comprehension, suggesting them source code descriptions by mining natural language descriptions extracted from developers communications. Chapter 7 presents an approach, named YODA (Young and newcOmer Developer Assistant) aimed at identifying and recommending mentors in software projects by mining data from mailing lists, issue trackers and versioning systems. Chapter 8 presents a recommender that mine messages exchanged among contributors/developers, in the form of issue trackers and emails, and extract useful descriptions, that describe specific source code elements.

Part I

Analysis of Developers' Communication

In this part of the thesis we are interested to understand what kind of information can be obtained by analyzing data from versioning systems and development discussions to support the newcomer with the *team work*. For this reason we analyze unstructured data form mailing lists, issue trackers and IRC chat to extract the social interactions between project developers. We also analyze versioning systems to extract important facts about the code changes applied by experienced developers. Hence, this part of the thesis reports three different studies.

The first study, reported in Chapter 2, investigates the usefulness of different communication channels, i.e., mailing lists, issue trackers, IRC chat, in recommending mentors and coordinators (developers having a high degree in the communication channels) in open source projects. The results suggest the usefulness of mailing lists and issue trackers in identifying coordinators and mentors in software projects, while other sources of information have too much informational noise to be precise and concise in this recommendation. This result suggests that mailing lists and issue trackers are reliable sources of information to mine developers collaborations and thus, built DSN.

For this reasons, in Chapter 3 we observe how developers contributing to open source projects spontaneously group into "emerging" teams, reflected by messages exchanged over mailing lists and issue trackers. We investigate whether, when a project evolves, emerging teams tend to re-organize themselves (e.g., by splitting or merging) reflecting events within project such us, releases of new software. Results of a case study show that such evolution of teams is reflected in cohesive changes in the software structure. Such information can serve to suggest source code re-modularization actions based on such "social information".

In Chapter 4 we study the evolution of the Apache (Java) ecosystem, in terms of number of developers, their interactions and the dependencies between projects and investigate (i) how dependencies between projects and the developers interactions evolve over time when the ecosystem grows; (ii) how developers discuss the needs and risks of such upgrades. The conjecture is that developers manage dependencies between projects relying on developers' communication channels. We discover that developers discussions are massively characterized by dependencies discussion of dependent sub-projects. Such information can be used to define an approach to avoid changes that could break the dependency with TPL.

Chapter 2

How Developers' Social Networks Built on Different Sources Differ

Contents

Software developers and, in particular, newcomers in software projects need to become confident with technical competencies and social skills to be productive and effective in the project. Thus, a proper newcomer training and a good project environment are highly desirable because impact the probability of newcomers to became a long term contributors [22]. The newcomers decision to abandon the project can be influenced by several factors [18, 22]. For example, as previous studies shown, there is a consequent low permanence rate of junior developers when they did not receive any answers (any support) by senior developers in the project [19, 22].

Developers communication channels, i.e., issue trackers, mailing lists and IRC chat are used by developers to exchange information about the project status and discuss development aspects. Thus, such sources are useful to extract relevant facts from the "social project environment". DSN mined from mailing lists can be analyzed to compute metrics to suggest the more active and more appropriate developers for the newcomers recuitment [27]. IRC chats that are used by developers to have on-line meeting and organize the work, reports information about the project status and the project discussions about specific and relevant development topics [28]. Thus, written communication recorded through channels such as mailing lists or issue trackers, but also code co-changes, have been used to identify emerging collaborations in software projects. However, it can often happen that some people tend to use some communication channels more than others. Also, collaborations identified through co-changes may differ from those identified through communication channels. Our idea is that different sources can tell different/complementary stories about DSN, for example, miss links between developers that are in the realty connected. Indeed, as pointed out by a work of Guzzi *et al.* [31], mailing list is only one of the communication channels, in OSS projects, as it also includes other channels such as the issue repository. For such reasons, in this Chapter, we report a study that analyze DSN built considering several sources and investigate how collaboration links vary and complement each other when they are identified through data from (i) three different kinds of communication channels, i.e., mailing lists, issue trackers, and IRC chat logs, and (ii) changes people performed on the same artifacts within a close time frame. Thus, the key goal of this Chapter is to understand what can be extracted/obtained by the various sources of information, and what are the commonalities between them. Results of a study reported in Section 2.3 over six open source projects indicate that the overlap of communication links between the various sources is relatively low and varies between projects. This means that, the identification of key project roles for project newcomers —e.g., high degree—lead to different results when using different sources.

2.1 Motivation: the importance of the "social environment" in OSS projects

The communication among projects' members plays a paramount role in any successful software project. In such a context, a good "social environments" for a project newcomer plays an important role in his/her permanence in the projects. Indeed, team coordination and communication has always been the crux of people involved in software project management [33]. Notwithstanding the nature of a project (*i.e.,* open source versus industrial/closed source), its domain, or size, the involved people need to exchange information effectively, minimizing the communication overhead and making sure they are up to date with the project status.

Analyse, such communication can be useful to identify key roles in DSN. Relevant roles such as, expert/mentors in open source project that can train project newcomers. However, people contributing to a project may prefer a particular communication channel. For example, general discussions about a project's perspective, software design, or future development strategies may happen in mailing lists, whereas discussions related to specific features or to the resolution of bugs occur on issue trackers. Another factor is the size, structure and general organization of the project. For example, some projects tend to have in the past most of the discussion over mailing lists, and only in recent years they tend to use issue trackers much more. Finally, in industrial projects part of the discussion occurs through face-to-face or phone meetings [34]. Thus, to identify correctly people that have key roles in a project it is important to select the appropriate "social sources information" depending on the project peculiarities.

In recent and past years, (written) communication has been analyzed by several authors for different purposes. For example, Bird *et al.* [35] and Hong *et al.* [36] studied to what extent emerging teams identified from email and issue tracker communication reflect the latent structure of software projects. Bettenburg *et al.* [37] and Kumar *et al.* [38] studied how social network metrics could be used for bug prediction purposes. Canfora *et al.* [11] used data from mailing lists and issue trackers to recommend mentors.

The studies mentioned above have analyzed projects' communication by observing one or two sources of communication. The conjecture we want to investigate is that, *by considering different sources (or their combinations), the view one can have of the collaboration network between project contributors change*.

To this aim, we analyze written communication between developers (*i.e.,* people changing the code) recorded through mailing lists, issue trackers, IRC chat logs, and code co-changes (the latter indicates that people got in touch if they modified the same artifacts in a limited time frame). The overarching goal is to provide evidence that by analyzing a single com-

Table 2.1: Characteristics of the analyzed projects.

Project	URL	Year Started	Observed Period	Size (KNLOC)	#Commits	#Comments in issue tracker	#E-mails in mailing list	#Messages in IRC chat
Apache CXF	http://cxf.apache.org	2005	June 2011-June 2013	593–771	4,911	6,016	3,049	305,802
Hibernate	http://hibernate.org	2003	June 2011-June 2013	984–1,096	1,805	992	2,423	84,218
Infinispan	http://infinispan.org	2009	June 2011-June 2013	146–286	2,482	9,305	3,886	893,780
Apache Lucene	http://lucene.apache.org	2000	June 2011-June 2013	198–437	2,957	68,055	10,821	104,901
Samba	http://www.samba.org	1996	June 2010-June 2012	1,278–1426	11,151	9,132	9,979	17,591
Weld	http://weld.cdi-spec.org	2008	June 2011-June 2013	108–139	1,225	1,996	2,423	98,044
Total	–	–	–	–	24,531	96,496	32,581	1,504,336

munication channel one may obtain a misleading portrait of people interaction, and that in general different combinations of the sources may provide different views of the project's interaction.

By analyzing the communications of six open source projects, namely CXF, HIBERNATE, INFINISPAN, LUCENE, SAMBA, and WELD we show that (i) not all developers use all communication sources; (ii) people interacting using a given channel may or may not communicate through a different channel; (iii) the identification of key project roles—such as developers with a high communication degree—leads to different results if done over different communication channels.

Section structure: Section 2.2 presents the details of the empirical study design, selected system, approach adopted to collect and analyze data. Section 2.3 reports empirical findings and is followed by Section 2.4 where we discuss the threats to validity. Section 2.5 discuss the related work while the Section 2.6 summaries the results of this study.

2.2 Empirical Study Design

The *goal* of the study is to analyze developers' collaborations mined from different sources of information with the *purpose* of understanding their commonalities and differences. The *perspective* is of researchers interested in studying to what extent using different sources could produce a different view of how developers interact in a project. When such a view is used to build different kinds of recommenders—*e.g.,* to suggest mentors—this could produce different results.

The *context* of the study consists of data from six open source projects, whose characteristics are summarized in Table 2.1: when the project started, the observed period, code size in non-commented KLOC (KNLOC), size of data from the four sources of information. CXF is a framework providing APIs for web service development while HIBERNATE is an object-relational mapping library for Java. INFINISPAN is a data grid platform written in Java and designed to be highly scalable. LUCENE is a Java-based indexing and search technology. SAMBA is a re-implementation of the SMB/CIFS networking protocol mostly written in C.

Finally, WELD is an implementation of the Contexts and Dependency Injection for Java EE. On the one hand, we have chosen such projects to ensure enough diversity in terms of size (of the code based, of the developers' population and of the exchanged messages). On the other hand, we looked for projects having the availability of data from the four investigated sources—versioning systems, issue trackers, mailing lists, and IRC logs—for a period of at least two years; we deemed two years long enough to observe collaborations.

2.2.1 Research Questions

In the context of the study, we formulated the following research questions:

- **RQ$_1$:** *To what extent do developers discuss through the different communication channels?* The conjecture is that some developers may use a limited set of the available communication channels. For instance, it may happen that only a small "core" team actually discusses through IRC, while many more may discuss over the issue trackers.
- **RQ$_2$:** *How do the inferred links between developers overlap when using different sources of information?* This research question investigates whether different sources of information provide a different view of the project social network or, in other words, of the project's members interactions.
- **RQ$_3$:** *What do the four different sources of information tell in terms of social network metric?* In this research question we analyze whether the identification of people having some particular role in the project, such as developers having a high degree in the communication change when using different sources of information.

2.2.2 Data Extraction Process

This section describes the data extraction process that we follow with the aim of collecting the data needed to perform our study.

Step 1: Data from Four Sources of Information

Commits checked in by developers are collected by mining the change log of the **versioning system** hosting the six subject projects. Note that the versioning system adopted for the analyzed systems, *i.e.,* Git, provides explicit information for authors, other than just for committers, although in many cases authors and committers match.

Issue trackers are mined with the aim of extracting developers' discussions carried out on this communication channel. In particular, for each system we download all issues cre-

ated in the analyzed time period (see Table 2.1) regardless their type (*e.g.,* bug, new feature, improvement) and status (*e.g.,* closed, open, resolved). To perform such a task we build two crawlers for the *Bugzilla* issue tracker (used by SAMBA) and *Jira* (used by the other five projects). For each issue, both crawlers extract (i) the name of the project member posting the issue, (ii) the issue title, (iii) the issue description, (iv) the posting date, and (v) the comments left by project's members to the issue, storing for each of them the name, the date, and the message. In total, we collected 5,173 issues comprising 96,496 comments.

Development mailing lists are downloaded from the Web, either by downloading available archives (*e.g.,*SAMBA, HIBERNATE, WELD, INFINISPAN), or by crawling Web-based mailing list (LUCENE and CXF). Then, emails are parsed to extract, for each message: (i) the message ID, (ii) the project's member sending the e-mail (*i.e.,* the *from* e-mail field), (iii) the project's member(s) to which the e-mail was sent (*i.e.,* the *to* e-mail field), (iv) the ID of the message being replied, (v) the e-mail subject, (vi) the email timestamp, and (vii) the message body. In total, 141,345 e-mails have been collected.

IRC chats are collected from the Web. In particular, for each discussion thread (reported in a separate page of the chat log) we store: (i) the (nick)name of developers taking part in the discussion, (ii) the thread date, and (iii) the messages exchanged in the thread. In total, 1,504,336 messages have been downloaded.

Step 2: Unifying Project Contributors' Names

We use an approach similar to the one used by Bird *et al.* [27] and used in our previous work [11]. The approach is composed of the following steps:

[1] **Normalization:** names are converted into lower cases, and special characters, including dots ".", are removed, e.g., *John F. Smith* becomes *john f smith*.

[2] **Ignore middle names**, *e.g.,john p Smith* corresponds to *john smith* unless this leads to an ambiguity.

[3] **First name referred with initials only**, *e.g.,john smith* corresponds to *j smith*, unless this generates an ambiguity.

[4] **Last name only**, *e.g.,john smith* corresponds to *smith* unless this generates an ambiguity.

[5] **Initials only**, *e.g.,j s* correspond to *john smith*, unless this generates an ambiguity.

[6] **User ID-like name:** IDs, often used in versioning composed by concatenating first and last names (or their initials). For example, *john f. smith* could be referred as *johnsmith*, *jsmith*, or *jfsmith*. Again, we check if the same ID can be obtained from multiple persons' names.

To deal with cases where email addresses are used in the project's members' communication, we use a set of heuristics, mostly derived from the above ones, to associate emails to names:

[1] **Extract name:** first, we extract the name from the email address, *i.e.,* anything preceding the "@" and split it into terms considering special characters as separators.

[2] **Map email address to a name:** we try to map the name extracted from the email to full names occurred in other emails. For example, *jsmith@google.com* is mapped to *John Smith*, even if he was previously associated with a completely different email address.

[3] **Map multiple email addresses of the same person:** we map multiple email addresses applying—on the name extracted from the email address—the same heuristics defined for names.

Overall, it is worthwhile to point out that the adopted approach for unifying names and email is a conservative one, *i.e.,* it performs an unification only when there are no multiple (ambiguous) possibilities of unification for the same name. Since we have no guarantee that the aforementioned approach is 100% accurate and complete, we integrated it with a manual analysis performed by two of the authors, aimed at verifying the existing mappings and adding missing ones. Such an analysis lasted four working days, and helped to fix less than 5% of wrong mappings and to add about 20% of missing ones.

Step 3: Extracting Developers' Links

Once unified the names, **we restrict our attention to commit authors'** only. This is because we want to focus our attention to discussions occurring between people involved in code changes only, rather than other people participating to the discussions.

Given two project's members, M_i and M_j, we identify a link $M_i \leftrightarrow M_j$ between them in the four sources of information by applying the following heuristics:

- **Versioning system:** M_i and M_j modify the same file during a specific time interval, fixed in this work to six months. Bear in mind this is not really communication, however it has been used in some past studies [39,40]. We considered the six months period as not so short (otherwise it would be unlikely to find links) nor so long that the two contributions were completely detached.
- **Issue tracker:** M_i and M_j left a comment to the same issue, M_i left a comment to an issue created by M_j (or *vice versa*).

- **Mailing list:** Both M_i and M_j sent emails / replied to the same email thread [27]. Emails belonging to the same thread have been identified by looking at the message ID of the email itself (for the email opening a thread) and the message ID of the email being replied.
- **IRC chat:** M_i and M_j take part to the same discussion thread.

2.2.3 Analysis Method

This subsection describes the analyses and statistical procedures used to address the three research questions formulated in Section 2.2.1.

To address $\mathbf{RQ}_1$, we compute and report the overlap (in percentage) of authors that used the various communication channels.

Similarly, for $\mathbf{RQ}_2$, we compute the overlap (in percentage) of links existing between different authors when considering different sources of information.

Besides such a quantitative analysis of the links, we are also interested to investigate the *nature* of the discussions occurring over the different communication channels. Undoubtedly, the most suitable way to do this kind of analysis is to rely on grounded theory, as done by Guzzi *et al.* [31]. However, This is not feasible when analyzing several sources from six projects. Instead, we perform two different kinds of analyses. First, we perform a quantitative analysis, done using topic models. For each project and for each communication channel, we build a topic model using Latent Dirichlet Allocation (LDA) [41]. LDA allows to fit a generative probabilistic model from the term occurrences in a corpus of documents. Basically each document is treated as a probability distribution of topics, in turn being distributions of words. The corpus for emails and issues consists of message subjects/bug title/short descriptions only (each of them represents a document in the corpus), because the rest of the message/issue discussion often contains details that would only add noise to the overall topic characterization. For IRC discussions, we took all messages (each of them is a document), since they are often very short and because no subject/short title is available in this case. The corpus is then processed by applying English stop word removal and Snowball stemming, and then topic models are generated. After, we analyze how topics discussed over the various communication channels are similar, by comparing the topic models using the Hellinger distance [42]. Given two discrete probability distributions P and Q, their Hellinger distance $H(P, Q)$ is:

$$H(P, Q) = 1/\sqrt{2}\sqrt{\sum_{i=1} k(\sqrt{p_i} - \sqrt{q_i})^2}$$

where k is the number of topics of the LDA model. Since the Hellinger distance varies between 0 and 1, and since we were interested to show the similarity of the discussion be-

tween pairs of communication channels, we convert it into a similarity as follows $S(P,Q) = 1 - H(P,Q)$. The whole topic analysis has been performed using the *topicmodels* package of the *R* statistical environment.

When applying LDA, one needs to calibrate the number of topics k, the smoothing factors for topic distributions in documents (α) and word distributions in topics (β), and the number of Gibbs iterations (n) required to generate the topic model. Although we are aware that LDA can produce sub-optimal results if not properly calibrated [43], in this case we did it by observing how our results vary by considering a number of topic $k \in \{25, 50, 100, 200\}$. For all projects, we did not notice any substantial difference when going beyond 50. For this reason, we have set $k = 50$. Similarly, we set $\alpha = 0.1$, and $\beta = 1/k$, and $n = 10$.

To address $\mathbf{RQ}_3$, we use the communication links extracted from the different sources of information to (i) recommend developers playing particular roles, and (ii) replicate the results of the study reported by Bird *et al.* [27]. Specifically, with respect to point (i) we rank developers using the following metrics:

- **Degree**: *i.e.,* the number of in-out communication links a developer has within a given communication channel [44]. The conjecture is that a person taking the leadership in a discussion would have a high degree. Degree metrics have been computed using the *R* package *sna*. To understand whether high-degree developers identified by the various communication networks are actually recognized as "important" developers by the community, we rely on the Ohloh[1] *Kudos* score. A *Kudos* depends on the level of appreciation or respect of a developer working for a project has, and it is based on the judgement of other project members[2]. Specifically, a member can give *Kudos* to other members, by assigning them a score ranging 1 and 10. An example of *Kudos* ranking for the project Apache CXF can be found at the URL `http://www.ohloh.net/p/cxf/contributors`.
- **Mentorship**: a project member is a mentor if s/he shows the ability to effectively train other people, generally newcomers. While the identification of developers with high degree is quite trivial, to identify mentors we rely on a recommender system described in the Chapter 7 and represents one of the main contributors of this thesis [11]. This approach is able to identify, given a newcomer joining the project in a given moment, the project's member that has been her/his mentor by taking into account factors like (i) the communication exchange between the newcomer and each project member, (ii) the level of sociability (degree) of each project member, and (iii) the difference in seniority of the newcomer with each project member.

[1] http://www.ohloh.net

[2] http://meta.ohloh.net/kudos

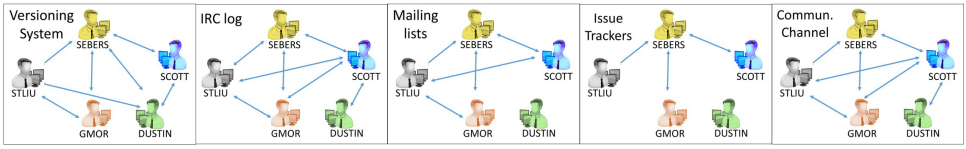


Figure 2.1: Hibernate: network of five developers as it is captured from different sources of information.

Note that the aforementioned *degree* and *mentorship* metrics are not Boolean, they rather indicate to what extent a project member plays (or not) one of the two roles described above.

2.3 Analysis of the Results

This section discusses the results achieved in our study and aimed at addressing the three research questions formulated in Section 2.2.1.

2.3.1 RQ_1: To what extent do developers discuss through the different communication channels?

Table 2.2 reports (i) the number of developers (*i.e.,* commit authors) contributing to the different sources of information; and (ii) the percentage of overlap between the different sources. In addition, we also considered the union of all communication channels (mails, issues, and chat). It is important to note that given a source S_i indicated on the row and a source S_j indicated on the column, the overlap of the authors $A(S_i)$ participating in S_i with the authors $A(S_j)$ participating in S_j is given by $|A(S_i) \cap A(S_j)|/|A(S_i)|$. For this reason Table 2.2 is not symmetric.

First, we can notice that a pretty good percentage of commit authors were found in the communication channels (column *cc*): such a percentage varies between 52% for WELD and 90% for SAMBA, with an average value of 75%. The communication channel attracting the largest percentage of authors varies between projects. For three projects (HIBERNATE, INFINISPAN, and CXF) the most popular channel is the chat, whereas for the other three (*i.e.,*SAMBA, LUCENE and WELD) it is the issue tracker.

In five projects out of six (*i.e.,* all but INFINISPAN), authors mainly use two out of three communication channels, whereas the third one is only used sporadically. For example, in Samba the issue tracker and the mails are used by 79% and 71% of the authors respectively, while only 21% use the chat. There may be many factors, such as the project size, internal organization and structure, or its age, that may influence the proneness of developers to use different communication sources. For example, SAMBA is relatively older than other projects

Table 2.2: RQ_1: overlap (in percentage) between authors contributing to different sources. $cc \equiv$ issues $\cup$ mails $\cup$ chat.

CXF					
	#authors	issues	chat	mails	cc
commits	21	57%	71%	43%	76%
issues	12		92%	58%	100%
chat	15	73%		53%	100%
mails	9	78%	88%		100%
cc	16	75%	94%	56%	
Hibernate					
	#authors	issues	chat	mails	cc
commits	77	24%	42%	34%	56%
issues	19		68%	68%	100%
chat	32	41%		59%	100%
mails	26	50%	73%		100%
cc	43	44%	74%	60%	
Infinispan					
	#authors	issues	chat	mails	cc
commits	40	73%	78%	75%	90%
issues	29		90%	83%	100%
chat	31	84%		87%	100%
mails	30	80%	90%		100%
cc	36	81%	86%	83%	
Lucene					
	#authors	issues	chat	mails	cc
commits	32	78%	53%	31%	84%
issues	25		64%	36%	100%
chat	17	94%		41%	100%
mails	10	90%	70%		100%
cc	27	92%	63%	37%	
Samba					
	#authors	issues	chat	mails	cc
commits	101	79%	21%	71%	90%
issues	80		25%	76%	100%
chat	21	95%		86%	100%
mails	72	84%	25%		100%
cc	91	88%	23%	79%	
Weld					
	#authors	issues	chat	mails	cc
commits	66	45%	32%	3%	52%
issues	30		56%	0%	100%
chat	21	81%		9%	100%
mails	2	0%	100%		100%
cc	34	88%	62%	5%	

(16 years of life, since 1998) and developers used for years mailing lists to communicate.

Only recently, they also adopted an issue tracker and, very recently, developers began to systematically use the chat. Indeed, the set of authors that exchange messages over issue tracker and mailing lists largely overlaps with the (small) set of people using the chat (95% and 86% respectively).

However, not all old projects have such a behavior. Consider, for example, LUCENE. This is a relatively old project (2000), however developers mainly rely on chat and issue trackers to exchange messages and organize their work. This also confirm what Guzzi *et al.* [31] found when analyzing its mailing lists. LUCENE, indeed, has differences from SAMBA in terms of number of authors (32 vs. 101) and domain (it is more a scientific project than a widely-used utility like SAMBA). In some sense, developers form a sort of "small community" that tend to gather a lot over the chat.

HIBERNATE is also different from the above projects, this is mainly because developers began to use IRC just two years after the project started. INFINISPAN is also a young project (2008), and in this case the use of all communication channels is very balanced: 73% for the issue tracker, 78% for the chat and 75% for emails. Last, but not least, WELD authors very rarely used emails during the observed period. That is, developers find more convenient to directly interact through chat or to discuss specific issues over proper means, *i.e.,* the issue tracker.

RQ_1 Summary: It is unlikely that all developers communicate over all channels, therefore to properly observe their interaction multiple channels should be considered. Also, while in the past developers used emails as main communication channel, nowadays they are massively using chats (more interactive) or issue trackers (better structured).

2.3.2 RQ_2: How do the inferred links between developers overlap when using different sources of information?

Table 2.3 reports the number of authors' links found in the different sources of information, and the overlap (in percentage) between the various sources (plus the union of all communication channels *cc*). A link represents a pair of authors that interact within a source. It can be noticed that the number of links identified by considering commits is relatively low, *i.e.,* only a subset of the authors worked on the same files in a time window of six months. In most cases, the sources exhibiting the highest number of links are issue trackers and chat logs; however this could be partially due to the fact that in this case all people participating to a discussion are considered linked.

Table 2.3: RQ_2: Number of author links found in the different sources of information, and overlap (in percentage) between them.

CXF						
	#links	commits	issues	chat	mails	cc
commits	73		14%	26%	5%	38%
issues	30	33%		40%	13%	100%
chat	85	22%	14%		2%	100%
mails	11	36%	36%	18%		100%
cc	109	26%	28%	78%	10%	
Hibernate						
	#links	commits	issues	chat	mails	cc
commits	184		3%	1%	12%	21%
issues	19	26%		16%	26%	100%
chat	248	8%	1%		13%	100%
mails	81	28%	6%	41%		100%
cc	307	13%	6%	81%	26%	
Infinispan						
	#links	commits	issues	chat	mails	cc
commits	193		33%	36%	22%	63%
issues	147	43%		37%	29%	100%
chat	445	16%	12%		19%	100%
mails	165	26%	25%	50%		100%
cc	593	20%	25%	75%	100%	
Lucene						
	#links	commits	issues	chat	mails	cc
commits	195		19%	11%	4%	27%
issues	140	27%		29%	4%	100%
chat	110	20%	37%		5%	100%
mails	23	30%	26%	22%		100%
cc	222	23%	63%	50%	10%	
Samba						
	#links	commits	issues	chat	mails	cc
commits	729		16%	2%	13%	27%
issues	360	33%		1%	18%	100%
chat	50	28%	10%		18%	100%
mails	313	30%	21%	3%		100%
cc	647	31%	56%	8%	48%	
Weld						
	#links	commits	issues	chat	mails	cc
commits	82		5%	16%	0%	18%
issues	24	17%		38%	0%	100%
chat	109	12%	8%		0%	100%
mails	0	0%	0%	0%		0%
cc	124	12%	19%	88%	0%	

Links observed in the commits have an overlap with other sources ranging between 0% (commits vs. emails in WELD) and 36% (commits and chat in INFINISPAN). Clearly, the former 0% is due to the limited participation of authors in WELD mailing lists as observed in $\mathbf{RQ}_1$. When computing the overlap in the opposite direction (other sources vs. commits), we can notice relatively high values for issues (43% INFINISPAN, 33% CXF and SAMBA, 27%

LUCENE, 26% HIBERNATE). If merging all communication channels, the link overlap of commits with other sources raises up, going from 18% on WELD up to 63% on INFINISPAN. This highlights the importance of analyzing more than one communication channel when building developers' collaboration networks.

As already observed in $\mathbf{RQ}_1$, results reported in Table 2.3 also confirm that authors generally limit their interactions to few communication channels. In CXF, we can notice that the overlap between chat and mails is very low (2%, whereas the opposite is 18%), while it raises up to 40% between issues and chat. In HIBERNATE and INFINISPAN the highest overlap is between mails and chat (41% and 50% respectively). This is interesting because in principle mails and chat seem to be two very different communication means, and also the way links are mined is different (point-to-point in emails, all participants in a discussion for chats). However, it should be noted that both are means to plan project activities, whether issue trackers are usually adopted to discuss specific issues instead. In LUCENE, both issue tracker and chat have a limited overlap with mailing lists (4% and 5%, whereas the reverse overlap is 26% and 22% respectively). Instead, the overlap between chat and issue tracker is 37% (reverse 29%). The overlap between links in issue tracker and chat is also relatively high in WELD (38%) where the reverse overlap is however low (8%), that is there are many links in the chat that do not appear in the issue tracker. Finally, as also mentioned in $\mathbf{RQ}_1$, SAMBA developers are less prone to use the chat, and this explains its limited overlap with mailing lists and issue tracker.

In summary, by looking at Table 2.3 it appears that different communication channels tell different stories in terms of developers' communication. Let us consider, for example, the subset of five HIBERNATE developers depicted in Figure 2.1. The figure shows five different networks built considering the five sources of information considered in the study (i.e., versioning system, IRC, mailing lists, issue trackers, and the union of the last three). When considering only a source of information, some links may be missing: for example *Dustin* performs commit with others, but he talks with them only on the chat.

As also noticed by Shihab *et al.* [28], IRC online meetings are often planned to answer questions related to common project topics, or for brainstorming. For example, during an IRC meeting a very active author of HIBERNATE wrote: *"is there a better way? dunno like I said this is brainstorming and I have not given lots of thought to these cases"*. Another author said: *"but we also need to create the attributes and values in the entity binding.."*. Topics that are also often discussed on the IRC are related to planning testing activities *"however a pure standalone test suite would make things easier..."*. Also, developers discuss how to prioritize activities on issues and whether or not to open issues on the issue tracker *"okay I think it is a bug and I'm going to create a jira first"*.

Table 2.4: Similarity measure of topics extracted from different communication channels.

	issues vs. mails	issues vs. chat	mails vs. chat
CXF	0.86	0.11	0.01
Hibernate	0.11	0.02	0.03
Infinispan	0.07	0.03	0.03
Lucene	0.08	0.3	0.02
Samba	0.06	0.02	0.02
Weld	0.11	0.04	0.03

By applying topics model as described in Section 2.2.3, the words describing the topic with the highest probability of chats (for HIBERNATE) are *test, fix, plan, project, unresolved, migration, integration, branch*. Instead, the topic with the highest probability for the issue tracker contains *fail, error, test, issue, broken, valid, wrong, delete, build, core*, while the emails have *test, build, valid, core, api, branch, fail, error, build, documentation, strategies*.

If one wants to answer the question "how similar are the topics discussed over different channels?", Table 2.4 reports the similarity (computed using the Hellinger distance) of all channel pairs. One can notice that values in the first column (issues vs. mails) are always higher than those in the other two columns, where issues and mails are compared with the chat. Among other cases, one can notice the very high similarity in CXF between issues and emails (0.86). For this project, we noticed that the top topics for issues and mails share several words such as *test, build, valid, core, fail, error, doc, strategies.*. Recently, developers are using issue trackers more and more as a valid alternative to mailing lists to discuss various kind of issues, not only related to specific bugs to fix or features to add/improve. Vice versa, the IRC chat has intrinsically a more interactive nature, and thus it is more suitable to brainstorming.

RQ_2 summary: The overlap of communication links between various sources is relatively low (generally below 30%-40%) and varies depending on the project. Therefore, data from multiple channels should be merged to have a better view of developers' interactions. The topics being discussed in issues and mails are closer to each other than those discussed in the IRC chat.

2.3.3 RQ_3: What do the four different sources of information tell in terms of social network metric?

Table 2.5 reports the percentages of overlap between the top five *coordinators* (authors having the highest degree in the communication or co-change with others) and *mentors* for the different sources of information. Note that we did not identify mentors from the commits as this does not make sense. In addition, always in Table 2.5 we also report for each source of

Table 2.5: RQ_3: Percentage of Overlap between Top Five *Coordinators* and *Mentors* as Extracted From the Four Sources of Information.

CXF	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mails	cc
commits	40%	20%	40%	20%	40%	-	-	-
issues		20%	40%	20%	20%	40%	60%	60%
chat			20%	100%	20%		20%	40%
mails				20%	**60%**			60%
Hibernate	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mail	cc
commits	40%	40%	40%	60%	20%	-	-	-
issues		40%	40%	20%	60%	20%	40%	40%
chat			40%	80%	40%		20%	20%
mails				60%	**60%**			60%
Infinispan	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mail	cc
commits	80%	40%	80%	80%	40%	-	-	-
issues		40%	80%	80%	40%	20%	60%	60%
chat			40%	60%	0%		20%	20%
mails				80%	**60%**			100%
Lucene	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mail	cc
commits	40%	0%	20%	80%	**60%**	-	-	-
issues		20%	0%	60%	40%	40%	20%	60%
chat			0%	20%	20%		20%	40%
mails				20%	**60%**			40%
Samba	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mail	cc
commits	60%	20%	80%	80%	**80%**	-	-	-
issues		0%	60%	60%	60%	20%	60%	80%
chat			20%	40%	20%		40%	40%
mails				80%	**80%**			80%
Weld	**[Coordinators]**					**[Mentors]**		
	issues	chat	mails	cc	kudos	chat	mail	cc
commits	60%	0%	0%	20%	0%	-	-	-
issues		0%	0%	20%	**20%**	40%	20%	60%
chat			0%	80%	0%		20%	60%
mails				0%	**20%**			40%

information, the percentages of overlap between the top five *high-degree* authors and the top five developers that obtained the highest *Kudos scores* (column *kudos* in Table 2.5).

In terms of *coordinators*, the percentage of overlap between the different sources is quite low (36%, on average). In 68% of the cases the overlap between the compared pairs of sources is $\leq 40\%$, and just in 20% of the cases the overlap is $\geq 80\%$. Vice versa, when recommending *mentors*, the average overlap between all pairs of sources is 41%, in 67% of cases the overlap is $\leq 40\%$, while just in 9% of cases it is $\geq 80\%$. However, as highlighted by

Table 2.6: RQ_3: Hibernate's Top five Project's Members: Coordinators and Mentors.

Coordinators						
Rank	commits	issues	chat	mails	cc	kudos
1	sebers	emmanuel	sanne	sebers	sebers	gavin
2	stliu	hardy	scott	sanne	sanne	sebers
3	lukasz	gmorling	gail	hardy	gail	emmanuel
4	bmeye	bmeye	emmanuel	emmanuel	scott	hardy
5	gail	sebers	sebers	stliu	stliu	erik
Mentors						
Rank	commits	issues	chat	mails	cc	
1	-	bmeye	bein	sebers	sebers	
2	-	hardy	pmui	emmanuel	scott	
3	-	lukasz	suppor	max	sanne	
4	-	emmanuel	adnan	hardy	hardy	
5	-	sanne	stuartdou	sanne	stliu	

the results of the RQ_2 topics (and links) discussed in mail are closer to the topics (and links) discussed in issue tracker. Thus, ideally, if we focus the attention between these two communication channels we expect to have an higher overlap in terms of coordinators and mentors. This result is confirmed in Table 2.5. In particular, the percentage of overlap of the coordinators between mail and issue is 40% on average and the percentage in terms of mentors is 47% on average. Vice versa, the chat obtained the lower overlap of mentors/coordinator with the other communication channels (it is very often lower than 20%). If we do not consider the chat, in terms of *coordinators*, the percentage of overlap between the different sources increase from 36% to 46% (in average), while in terms of *mentors*, the percentage of overlap between the different sources increase from 41% to 47% (in average). Thus, it is clear that the chat channel identifies a set of mentors/coordinators that are very decoupled with the set of mentors/coordinators provided by the other communication channel. The overlap of the top *Kudos* developers with those having the highest degree highlights such a finding. By looking Table 2.5, is evident that the lowest overlap between top *degree* and top *Kudos* is obtained by the chat channel, while the highest overlap is achieved by the mail. This means that, by computing high *degree* on the network obtained from mails, we are able to identify developers that have a high reputation in the project.

In summary, recommendations of coordinators and mentors computed using developers collaboration networks mined from different sources can be quite different. The set of mentors/high-degree developers identified relying on chat is very decoupled with the set of mentors and high degree developers identified by the other channels. This analysis also finds confirmation in the analysis of *Kudos*.

Table 2.6 reports the top five *coordinators* and *mentors* extracted from each source of information of HIBERNATE. If one is interested in knowing which are the *coordinators* of the

HIBERNATE project, she could choose to mine any of the available sources of information, achieving however different results case by case. Indeed, the project's author *sebers* is the only one retrieved as coordinator in all cases, while substantial differences can be observed for other authors. An interesting case is related to the HIBERNATE developer *sanne*, that is identified as a coordinator when mining chat, mails, or the union of all communication channels (*cc*), while he is not in the top five when mining commits (he is a committer, nevertheless) and issues. His Linkedin profile[3] mentions that he is one of the project's members leading HIBERNATE. However, if for instance one limits the collaboration/communication analysis to commits and issues, this information would not emerge.

RQ_3 summary: Social network studies and recommenders in software engineering should not limit their information mining to a single sources. However, some social network metrics extracted from the different sources may have a different interpretation, e.g. high degree on chat does not necessarily correspond to high code change activity.

2.4 Threats to Validity

Construct validity threats concern the relationship between theory and observation. Such threats are mainly due to imprecision in the mapping of names used in different sources, and in how links were identified. As for the unification/mapping of names, as explained in Section 2.2.2 we have used an approach inspired from previous work [11, 27, 45] and complemented it by a thorough manual validation. However, we cannot exclude possible mistakes. Nevertheless, given the high number of developers involved in the study, it is unlikely that small deviations will change the essence of our findings. Concerning the identification of links, we used state-of-the-art approaches to identify links in mailing lists, issue trackers and chats. However, we are aware that the participation to an issue in issue trackers does not mean communicating with everybody involved there, and similarly it is likely that not everybody in a chat session is really involved in each specific discussion. Finally, we are aware that links inferred from versioning system may have little value because people working on the same file might never get in touch. Nevertheless, our aim is to show that links extracted from code changes or from communication channels, although overlapped, have different meaning and therefore can be quite different.

Threats to *internal validity* concern factors that could have influenced our results. Our study is based on what in our opinion are the most widely used communication channels in open source projects. Other channels—*e.g.,* microblogging through Twitter—indeed exist.

[3] `http://it.linkedin.com/in/sannegrinovero`

Consistently with a very recent work by [46] we found that for the analyzed projects that "Twitter doesn't allow for long discussions (developers prefer to use other channels for that purpose)", thus, it is mainly used for advertisement purposes. Last, but not least, besides all (written) sources of information one can consider, we are aware that there is still a portion of the developers' communication happening by voice, and that are not traceable elsewhere [34].

External validity threats concern the generalization of our results. The study is limited to six systems and, for consistency and comparison between projects, to the most recent project years. Although we expect that similar findings will be obtained if other projects are analyzed we cannot be sure, further, larger studies need to be conducted to generalize and confirm (or contradict) our findings.

2.5 Related Work

In the following, we discuss related work concerning the analysis of developers collaboration networks for various purposes in the context of software engineering (SE) studies, and with the aim of building software engineering recommender. The main aim here is to show—as also summarized in Table 2.7—that different work used different sources of information.

Previous studies analyzed DSN applying Social Network Analysis (SNA) on data extracted from Versioning Systems repositories [39,40,47–53]. For example, Xu *et al.* [52] used SNA to study the developers community at SourceForge, finding that the obtained developer network is a scale-free network. Pohl *et al.* [40] and Linguo *et al.* [53] showed how social networks could be used to determine roles in the community of developers belonging to the a software project. We share with Pohl *et al.* the approach used to identify relations between developers from versioning system data (two developers are connected if contributed to the same file during the same period). Studies by Singh *et al.* [50] and Lopez *et al.* [48] observed how committers networks is small-world network. Moreover, projects where there are small clusters of developers that discuss with each others are likely to exhibit success [50]. Surian *et al.* [51] findings are consistent with those of Singh *et al.* [50]; that is, the small-world phenomenon also exists in SourceForge, especially when developers in a network are separated, on average, by approximately 6 hops. More recently, Meneely *et al.* [49] used two issue tracking annotations—*i.e.,* solution originator and solution approver—from bug database to complement the developers network of versioning data. We share with this work the importance of using multiple sources of information. In a subsequent work, Meneely *et al.* [39] empirically showed that SNA metrics represent socio-technical relationships in open source development projects. This reflects the work done in our $\mathbf{RQ}_3$, which however highlights that such socio-technical relationships may change when using different sources of information.

Various authors have investigated developers' collaboration through mailing lists [27, 31, 54–56]. Bird *et al.* [27] discovered that—in mailing list DSN—few members account for a large proportion of messages sent and of replies. They also found high correlations between various social network status metrics and source code development. The latter finding was also confirmed by Shihab *et al.* [55]. Bird *et al.* [35] analyzed the relationship between communications structure and code modularity, and found that sub communities identified using communication information are related to code collaboration behavior. The heterogeneity of email content and discussion was investigated by Bacchelli *et al.* [54] and Guzzi *et al.* [31]. Bacchelli *et al.* [54] presented a technique that classifies email lines into five categories (text, junk, code, patch, and stack trace) and evaluated such approach on a (statistically) significant amount of emails gathered from mailing lists of four unrelated open source systems. Guzzi *et al.* [31] quantitatively and qualitatively analyzed a sample of 506 email threads from the development mailing list of Apache Lucene. Their study shows that developers participate in less than 75% of the threads, and that in only about 35% of the threads source code details are discussed.

Hence, very likely as it was also found in our study, developers also discuss through other communication channels, including issue trackers and IRC. Indeed, IRC meetings are increasing in popularity among OSS developers [57]. Elliot *et al.* [58] reveal how, use IRC instant messaging streams, persistent IRC logs and mailing lists help not only to build a community but also resolve conflicts. Shihab *et al.* [28] analyzed IRC logs and found that (i) a small and stable number of the participants contribute the majority of messages in meetings, (ii) there are common discussed topics as well as project specific topics. LaToza *et al.* [59] surveyed eleven developers with the aim at investigate common practices and their satisfaction in software development. They discovered several barriers preventing email (and in general written communication) usage. They found that face-to-face communication have advantages. Also the found that the use of more interactive communication channels (like IRC) is more desirable than emails.

While mailing lists were used a lot in the past, nowadays many projects are moving most of the discussion onto issue trackers, that are used besides the simple discussion of bugs to be fixed. For this reason, various authors have proposed developers based on issue trackers. Haythornthwaite [62] found that the set of core developers identified considering interactions on issue trackers differ from the "formal" lists of contributors published on projects' Website. Hong *et al.* [36] compared the evolution of DSN extracted from issue trackers with evolution of general social networks (*e.g.,* Facebook or Twitter etc.), finding some commonalities and differences. Kumar *et al.* [38] investigated the impact of global social network properties on the bug fixing process. Their results, suggest that the higher the average degree of a

Table 2.7: Studies that analyzed Developers Social Networks.

Source of information	Related papers
Mailing lists	Bacchelli *et al.* [54], Bird *et al.* [27] Wagstrom *et al.* [56], Guzzi *et al.* [31] Bird *et al.* [35], Shihab *et al.* [55]
Issue Trackers	Bernardi *et al.* [60], Crowston *et al.* [61] Zhou *et al.* [22], Haythornthwaite [62] Hong *et al.* [36], Kumar *et al.* [38] Bettenburg *et al.* [37]
IRC channel	Elliot*et al.* [58], Shihab *et al.* [57] Shihab *et al.* [28]
Microblogging (*e.g.,* Twitter)	Zhao *et al.* [63], Zhang *et al.* [64] Ehrlich *et al.* [29], Dullemond *et al.* [65]
Versioning Sistems	Lopez *et al.* [48], Meneely *et al.* [39] Capiluppi *et al.* [47], Pohl *et al.* [40] Singh *et al.* [50], Surian *et al.* [51] Xu *et al.* [52], Linguo *et al.* [53]
Mailing lists and Issue Trackers	Begel *et al.* [66]
Issue Trackers and Versioning Systems	Meneely *et al.* [49]

communication between developers, the lower the average time to fix a bug. Similarly to Kumar *et al.* [38], Bettenburg *et al.* [37] find that the consistency of the communication flow correlates with fault-proneness. Other works by Crowston *et al.* [61] and Zhou *et al.* [22] used co-occurrence of developers on bug reports as indicators of a social link. Crowston *et al.* [61] observed that development teams vary widely in their communications centralization, from projects completely centered around one developer to projects that are highly decentralized. With the aim at address the problem of inter-team coordination Begel *et al.* [66] presented Codebook, a framework for connecting engineers and their work artifacts together.

Recently, several researchers investigated and evaluated the the role played by communications in Twitter and more in general, the role played by "microblogging", in software development organizations [29, 63–65]. Zhao *et al.* [63] surveyed 11 microblog participants to better understand the conversational aspects of Twitter discovering the potential benefits it brings to informal communication at work. However, as Zhang *et al.* [64] highlighted, there is a large variation in the posting activity of various users, and there are *barriers* in adopting such new social communication channels. Moreover, Ehrlich *et al.* [29] showed how different are the use of the external/internal microblogs: external microblogs are used for share

general information; instead, internal microblogs are used to technical assistance and discussion. Finally, Dullemond *et al.* [65] evaluated microblogging discussions, and found how (i) sustaining a higher feeling of connectedness with people geographically distributed, (ii) where "mood-activity environment" helps to obtain information that are traditionally harder to obtain in a less volatile form. In summary, although there are barriers, microblogging could likely become another promising communication channel. However, we did not consider it in our study, because (i) we found that the Twitter accounts of the analyzed projects are mainly used for advertisements, *e.g.,* of new releases; (i) since we deal with (sometimes large) open source projects rather than close organizations, it is not feasible to keep track of the Twitter accounts of all developers (if any).

2.6 Summary

In this study we analyzed developers' communication over different channels (mailing lists, issue trackers, IRC chat) and their co-change activities captured from versioning systems. The study concerned a period of observation of at least two years for six open source projects, namely CXF, HIBERNATE, INFINISPAN, LUCENE, SAMBA, and WELD. Results of the study highlighted that (i) not all developers use all communication channels; (ii) people interacting through a given channel may not necessarily also communicate through other channels; and (iii) the identification of key project roles—such as developers with a high communication degree or mentors—leads to different results if done over different communication channels.

In summary, results tell that analyzing developers collaboration/communication through specific channels would only provide a partial view of the reality. Thus, a newcomer should relying in more than communication channel, to have a real indication of the social/technical roles played by a given developer. Therefore, if using specific collaboration/communication networks for various purposes—*e.g.,* identifying experts—one should be careful as results may not completely reflect the reality. We find that issue trackers and mailing lists are more appropriate sources to identify coordinators with respect to the chat.

This is also because different channels are likely to be used for different purposes: issue trackers document bug-fixing activities, but also enhancement and feature requests, that are often also discussed in development mailing list. A more interactive discussion often occurs through chats, *e.g.,* with the aim of planning activities such as testing or prioritizing issues. Finally, one must be aware that some discussions still occur by voice and face-to-face meetings [34, 59].

Chapter 3

Evolution of Emerging Collaborations and its Relation with Code Changes

Contents

In the previous Chapter we analyzed DSN built considering several sources and investigated how collaboration links vary and complement each other when they are identified through data from (i) three different kinds of communication channels, such as, mailing lists, issue trackers, and IRC chat logs, and (ii) changes people performed on the same artifacts within a close time frame. The obtained results (a study reported in Section 2.3) over six open source projects indicate that the overlap of communication links between the various sources is relatively low and varies between projects. This means that, the identification of key project roles for project newcomers —e.g., high degree—lead to different results when using different sources. It is important to note that we focused our attention on the social interactions between developers of a small projects. However, we don't know how developers interactions change when a project is growing in terms of number of developers and number of sub-projects. When such growing in a OSS project involve an higher number of dependencies between sub-projects, as well as, between the developers sub-communities, this evolution of an OSS take the name Software ecosystem.

However, what we do not already observed is how developers contributing to open source projects and spontaneously group into "emerging" teams, reflected by messages exchanged over mailing lists, issue trackers and other communication means. Previous studies suggested that such teams somewhat mirror the software modularity. This Chapter empirically investigates how, when a project evolves, emerging teams re-organize themselves—e.g., by splitting or merging. We relate the evolution of teams to the files they change, to investigate whether teams split to work on cohesive groups of files. Results of this study—conducted on the evolution history of four open source projects, namely Apache HTTPD, Eclipse JDT, Netbeans, and Samba—provide indications of what happens in the project when teams reorganize. Specifically, we found that emerging team mergers and splits working on more cohesive groups of files. Such indications serve to better understand the evolution of a software project by project newcomer. More important, the observation of how emerging teams change can serve to suggest software remodularization/re-factoring actions for newcomers/senior developers that are interested to better restructure the software components. Specifically, in our previous work [15] we present a technique to suggest refactoring based on team co-maintenance patterns. If two teams work on the same module (package, class, method), then the tool can suggest that this module can be refactored based on the activity patterns. We evaluate the approach using five projects from the Android API. Differently from this work we want to refactor modules identifying teams relying on social links extracted from mailing lists and issue trackers data.

3.1 Motivation: how project evolves and emerging teams re-organize themselves?

The organization of developers into teams is crucial for the success of software projects. In industrial projects, teams are often defined and staffed by project managers, that group people based on the needs of a particular task, and on people availability, skills, and attitude to work together. In open source projects it is still true because developers geographically distributed around the world organize their work in a similar way, relying often on electronic communication channels, such as mailing lists, issue trackers, IRC chat (as explained in Chapter 2).

In such context, for a newcomer that arrived in a software project having information about these socio-technical dynamics can play an important role in his/her permanence. Dynamics are different in open source projects [26], that involve developers spread across the world and working in different time zones, often communicating using electronic means such as mailing lists. In essence, developers participating in open source projects are not staffed into teams by project managers. Moreover, the way they collaborate depends on the structure of the open source project, i.e., whether the project is of "cathedral" or "bazaar" type [67]. Generally speaking, developers spontaneously group themselves into *"emerging teams"*, that can be recognized by observing the developers' communication network and how developers change source code files. In the rest of the following sections, we use such a definition of team.

Bird *et al.* [68] analyzed social networks and found that there is a causal consequence between the modularity of a software project and the way developers group into teams. They also found that developers belonging to the same sub-community share a larger proportion of files than developers belonging to different sub-communities.

When a software project evolves, the way emerging teams are formed and operate may change. This is because during its lifetime a project undergoes different kinds of changes, requiring the contribution of different, and possibly new, people. As pointed out by Hong *et al.* [36], emerging teams reorganization often happens in correspondence to new project releases.

Stemming from the above considerations, this study investigates how emerging teams evolve in open source software projects as people focus on different technical activities, i.e., code-changes. By analyzing how people collaborate through mailing lists and issue trackers, and what files they modify, we investigate *whether emerging teams evolve with the aim of working on more cohesive groups of files*. Figure 3.1 provides an overview of our analyses: on the one side, we identify emerging teams in different time periods following software

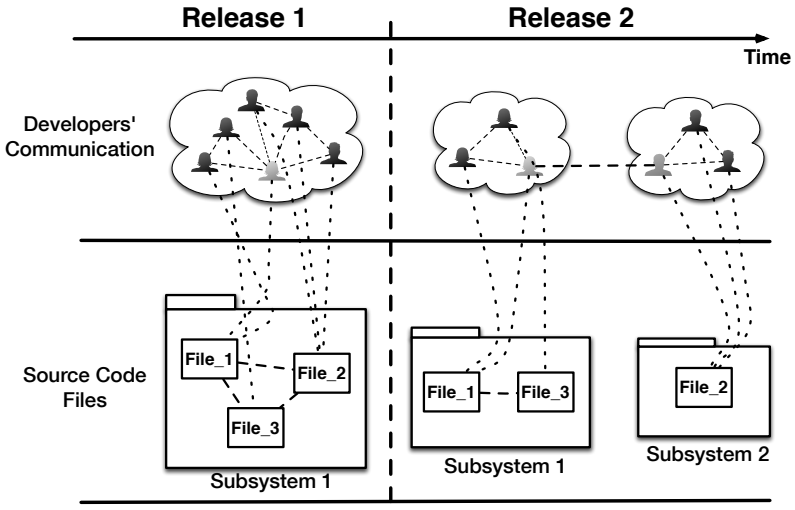


Figure 3.1: Evolution of emerging teams and of their technical activities.

releases, and map teams related to subsequent releases/periods. Then, we analyze what files these teams change in the versioning system, and analyze how the cohesiveness of such files changes when teams split or merge.

The study has been conducted on the evolution history—consisting of data from versioning systems, mailing lists and issue tracker—of four open source projects, namely Apache httpd[1], Eclipse JDT[2], Netbeans[3], and Samba[4]. We considered only, issue trackers an mailing lists as sources of information because, as observed in Chapter 2, other communication channels, like the IRC chat, are less precise and reliable in suggesting social connections between developers.

Results of the study indicate that, when emerging teams merge and split, we observe a significant increase of the cohesiveness—measured both structurally and semantically—of files on which such teams work. In other words, developers reorganize themselves in order to work on more cohesive groups of files.

The contributions of this study can be exploited in different ways. On the one hand, the natural (re)grouping of people in the project communication might suggest the need to re-structure the software design and/or source code organization. On the other hand, the observation of how the project communication evolve with respect to the source code structure

[1] `http://httpd.apache.org`
[2] `http://www.eclipse.org/jdt`
[3] `https://netbeans.org`
[4] `www.samba.org`

could help project leaders to better monitor and understand the project activities, as well as to provide them recommendations on how and when restaffing project teams. Such approach can inspire similar analysis being applied in an industrial context.

Structure of the Chapter. Section 3.2 provides the empirical study definition, formulates the research questions the study is going to address, and details the data extraction and analysis method. Section 3.3 reports and discusses the empirical study results. Threats to validity are discussed in Section 3.4. Finally, Section 3.6 summaries the results of this study.

3.2 Study Definition and Planning

The *goal* of this study is to investigate the evolution over time of emerging teams of developers in open source software projects. The *purpose* is to understand how such evolution relates to the cohesiveness of files people change over time. The *perspective* of the study is of researchers interested to observe the relationship between developers' communication and activities in software projects. Such observation can be used to devise recommenders for software project managers.

The *context* consists of the source code history, mailing lists and issue trackers of four open source projects, namely Apache httpd, Eclipse JDT, Netbeans, and Samba. Apache httpd (in the following abbreviated as Apache) is an open-source HTTP server, Eclipse-JDT and Netbeans are Java Integrated Development Environments (IDE), and Samba a cross-operating system layer for printer and file sharing. We have chosen these four projects to have a reasonable variety in terms of size and application domains: two of them, Apache httpd and Samba, are network applications, while the other two are really the same kind of system, i.e. Java IDE. The latter choice is also motivated to allow comparing projects that are pretty similar in terms of features offered. Also, we had to choose projects for which both mailing lists and issue trackers were available. Table 3.1 reports key information about the four projects and, above all, of their mailing lists, issue trackers, and versioning systems, namely project URL, time period analyzed, size range in KLOC, list of considered releases, number of mailing list contributors (Mc), number of issue tracker contributors (Ic), and number of authors indicated in the versioning systems (Au). Note that the versioning system adopted for the analyzed systems, i.e. Git, provides explicit information for authors, other than just for committers, although in many cases authors and committers match. In addition, the table reports the intersections $Mc \cap Au$, $Ic \cap Au$ and $(Mc \cup Ic) \cap Au$. Note that the latter is of particular interest for our study, in that it tells us how many authors communicated through the two communication means considered. In particular, the percentage of the authors that participated in discussions in mailing lists and/or issue trackers range between 68% and 80%.

As the work of Guzzi *et al.* [69] pointed out, sometimes core developers' participation in mailing lists is very low. In our cases, it is of less than 36% of authors for Eclipse JDT and Netbeans. For this reason, we decided to consider in our study both sources of information.

3.2.1 Research Questions

The study aims at addressing the following research questions:

- **RQ$_1$:** *How do emerging collaborations change across software releases?* This research question has a merely exploratory nature, and poses the basis for the subsequent one. It aims at observing how emerging teams evolve, i.e., to what extent these teams merge, split, and recombine. Also, we analyze the proportion of developers that join the project, those that are inactive in a given time period, and those leaving the project, to understand whether different projects exhibit different dynamics in terms of team reorganization.

- **RQ$_2$:** *How does the evolution of emerging collaboration relate to the cohesiveness of files changed by emerging teams?* This is the core research question of the study, and aims at investigating whether a team split or merger is reflected by change activity being done in more (or less) cohesive groups of files. Previous work [70] suggested that such a structure reflects the system architecture. Such information can provide multiple indications. If a team works on a set of unrelated files, in some circumstances it could be useful to split a team into smaller teams working on cohesive files.

  As it will be detailed in Section 3.2.2, we measure the cohesiveness of files modified by emerging teams using both structural and semantic measures, which have been found to capture different, complementary aspects of software modularity [71].

3.2.2 Data Extraction Process

To address these research questions, we need two data sources: (i) *developers' communication* (extracted from mailing lists and issue trackers), that we use to identify emerging collaboration teams; and (ii) *change history*, extracted from versioning systems, that is used to determine which files developers work on over time. In the following, we detail how such data sources are used to address the two research questions.

Table 3.1: Characteristics of the four projects under study.

VARIABLE	APACHE HTTPD	ECLIPSE JDT	NETBEANS	SAMBA
Period analyzed	09/1998-03/2012	01/2002-12/2012	01/2001-08/2012	01/2000-12/2011
Size range (KNLOC)	77-1,550	83-2,082	71-9,746	156-1,416
Releases considered	2.0 2.2.0 2.2.4 2.2.12 2.4.1	3.0 3.2 3.4 3.6 4.2	3.4 3.6 5.5 6.9 7.2	2.3 3.0.20 3.0.25 3.5.0 4.0
Mailing list contribs (Mc)	2,598	127	3,928	3,211
Issue tracker contribs (Ic)	5,727	2977	3,095	4,974
Committers (Au)	87	56	328	122
$Mc \cap Au$	64	11	118	69
$Ic \cap Au$	6	34	168	78
$(Mc \cup Ic) \cap Au$	70	38	218	96
Emails exchanged by $Mc \cap Au$	17,650	432	7,424	23,613
Bugs reported/discussed by $Ic \cap Au$	5,602	5,656	1,5365	14,262

Step 1: Extracting the communication network

To analyze the evolution of the emerging teams, we build communication graphs using data from mailing lists and issue trackers as described in Section 2.2.2.

We are interested to study the evolution of teams, therefore we build communication networks related to different time periods, corresponding to the intervals between two subsequent releases. As in some cases releases are frequently issued, analyzing only few months of communication would not be representative of the emerging groups. Hence, we analyze the period between one release and the next one such that the time interval between them is of at least one year. In the following, we refer to r_k the network concerning the communication between release r_k and release r_{k+1}.

Step 2: Identifying emerging teams

To identify emerging teams, we apply clustering algorithms on the social networks extracted in *Step 1*, using the intensity of the communication between two developers—i.e., the number of emails exchanged—as an indicator of the connection's strength.

To cluster developers, it is important to choose a suitable clustering algorithm, and, if the algorithm requires it, to determine the number of clusters. Previous work [72] used k-means as clustering algorithm. However, one issue of k-means is that it does not allow one item to be placed in more than one cluster. However, when identifying emerging teams, it is realistic to consider that a developer may primarily work for a team, but that she can also contribute to

other teams. To this aim, we use a fuzzy alternative to k-means, i.e., the fuzzy C-Means [73] which, other than assigning a developer to a team, also provides the *membership score* $ms_{i,j}$, i.e., the likelihood that a developer d_i can belong to a team t_j. Specifically, we used the *cmeans* function of the *e1071* package available in the *R* [74] statistical environment.

After having analyzed the distributions of likelihoods we need to identify which teams the developer contributes to. To this aim, we define a threshold, named *Membership Score Threshold* (MST). If $ms_{i,j} \geq MST$, then we assume that d_i contributes to t_j, while we consider the contribution negligible if $ms_{i,j} < MST$. In addition, we distinguish the primary team of a developer (the one with the highest ms_i) from the other teams to which she contributes.

In our study, we analyzed the values of $ms_{i,j}$ for all four projects and found that it is above 0.5 for the primary teams, whereas it is very low (and always < 0.3) for teams where a developer had an occasional or negligible participation. In conclusion, observation of such values suggested us to set MST equal to 0.3.

To determine the number of clusters k, we compute the Silhouette coefficient introduced by Kaufman and Russeeuw [75]. Let us consider a clustering obtained for a given k. For the observation i (in our case a developer) let $a(i)$ be the average distance to the other points (developers) in its cluster (the team), and $b(i)$ the average distance to points in the nearest cluster (a different team). Then the Silhouette statistics is defined as:

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))}$$

For different values of k, we will obtain different values of $s(i)$. Kaufman and Russeeuw suggested choosing the optimal number of clusters as the value maximizing the average $s(i)$ over the dataset. In literature, it is assumed that the Silhouette curve knee indicates the appropriate number of clusters [76]. In our study, we obtained $k = 10$ for Apache httpd and for Samba, $k = 7$ for Eclipse JDT and $k = 11$ for Netbeans. It is important to note that the different projects obtain different values of k because the average distance between developers (points) and in a team (cluster), as well as, and the average distance to developers (points) in the nearest team (cluster) varying between projects. Such values depend from the number of developers and the density of the DSN built considering the messages exchanged between the developers. After performing clustering, we discarded singleton groups. For this reason, in some cases our results can report a number of teams smaller than the k defined above.

Step 3: Mapping teams across different time intervals

After having identified the emerging teams, we compare the teams identified in two sub-

sequent time intervals r_k and r_{k+1}, and try to build a mapping between them. Specifically:

- a team t_j of r_k *splits* if two or more teams of r_{k+1} contain subgroups of t_j. We define a *subgroup* as a set of at least two developers belonging to a group. On the one hand, we did not impose a threshold greater than two because, as it will be clearer from the results, groups are often small. Also, this will allow us to observe how small sub-communities move. As it will be shown in Section 2.3, this frequently occurs in smaller projects. On the other hand, we did not consider singletons moving from a group to another as a case of group splitting.
- vice versa, we detect a *merger* if subgroups of two teams t_j and t'_j of r_k belong to the same team in r_{k+1}.
- finally, one team *survives* between r_k and r_{k+1} if at least one of its subgroups remains in r_{k+1} and no other subgroup merges with it.

Step 4: Analyze changes performed by the emerging teams

This steps analyzes to what extent developers belonging to the same emerging team work on a cohesive set of files. As explained before, we observe such a cohesiveness from both a structural perspective and conceptual (semantic) perspective.

As a structural measure, we rely on the *Modularity Quality (MQ)* defined by Mancoridis *et al.* [77]. *MQ* has been widely used in the context of software remodularization, for example as a fitness function for search-based remodularization algorithms [78].

Formally, *MQ* is defined as follows:

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j} & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases}$$

where intra-connectivity measurement A_i of cluster *i* consisting of N_i components and μ_i intra-edge dependencies is $A_i = \frac{\mu_i}{N_i^2}$. Moreover, $E_{i,j}$ is the inter-connectivity between the i^{th} and j^{th} clusters. Formally, the inter-connectivity $E_{i,j}$ between clusters i and j consisting of N_i and N_j components, respectively, with $\alpha_{i,j}$ inter-edge dependencies is defined as:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\alpha_{i,j}}{2N_iN_j} & \text{if } i \neq j \end{cases}$$

In our context, we compute the *MQ* for an emerging team by considering the set of files modified by the team members. As also done in previous studies [78], we assume a correspondence between modules and system directories. That is, Figure3.2 illustrates an example

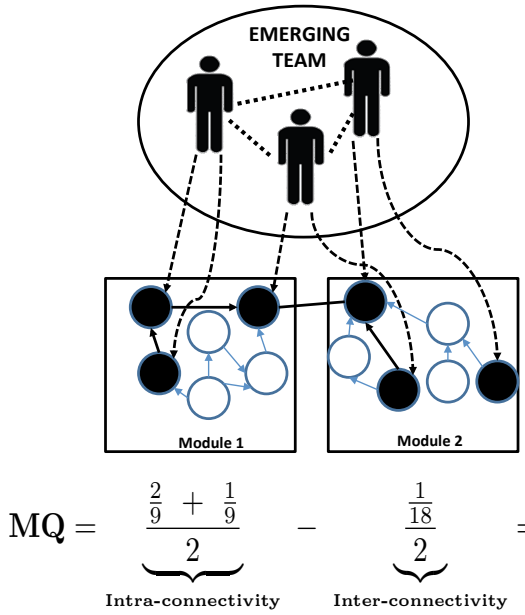


Figure 3.2: Modularization Quality (MQ) computation for files modified by an emerging team.

calculation of *MQ* for an emerging team of three developers, which changed files belonging to two different modules.

After having computed the *MQ*, we observe how it changes when teams split and merge over the observed evolution period.

Note that previous work by Bird *et al.* [68] measured the cohesiveness of files modified by an emerging team using *activity focus* instead of *MQ*. The activity focus [68] is defined as the average directory distance between files changed by developers belonging to the same team, and compared it with the average directory distance of files changed by developers belonging to different teams. We use *MQ* because, differently from activity focus, it also accounts for dependencies between files (besides how they are organized in directories). Nevertheless, we also checked how activity focus changes when teams split and merge, and found consistent results with *MQ*.

As a complement to the *MQ*, we use the Conceptual Coupling Between Classes (*CCBC*) defined by Poshyvanyk *et al.* [79]. The *CCBC* is based on the semantic information captured in the code by comments and identifiers. That is, two classes are conceptually related if their (domain) semantics are similar, i.e., they have similar responsibilities. The definition of *CCBC* requires the introduction of a lower-level measure [79]: the Conceptual Coupling Between Methods (*CCM*). To measure *CCM*, Latent Semantic Indexing (LSI) is used to

represent each method as a real-valued vector that spans a space defined by the vocabulary extracted from the code. The conceptual coupling between two methods m_i and m_j is then calculated as the cosine of the angle between their corresponding vectors [80]:

$$CCM(m_i, m_j) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{\|\overrightarrow{m_i}\| \cdot \|\overrightarrow{m_j}\|}$$

where $\overrightarrow{m_i}$ and $\overrightarrow{m_j}$ are the vectors corresponding to the methods m_i and m_j, respectively, and $\|\overrightarrow{x}\|$ represents the Euclidean norm of the vector x [80]. Thus, the higher the value of *CCM* the higher the similarity between two methods. Clearly, *CCM* depends on the consistency of naming used in the source code and comments.

Now we can define the conceptual coupling between two classes c_i and c_j as:

$$CCBC(c_i, c_j) = \frac{\sum_{m_h \in c_i} \sum_{m_k \in c_j} CCM(m_h, m_k)}{|c_i| \times |c_j|}$$

where $|c_i|$ ($|c_j|$) is the number of methods in c_i (c_j). Thus, $CCBC(c_i, c_j)$ is the average of the coupling between all unordered pairs of methods from class c_i and class c_j. The definition of this measure ensures that CCBC is symmetrical, i.e.,

$$CCBC(c_i, c_j) = CCBC(c_j, c_i).$$

It is worthwhile to point out that, while for structural modularity we relied on a measure capturing both cohesion and coupling information (*MQ*), for semantic modularity we only relied on coupling. This is because semantic cohesion and coupling measures provide a similarity indicator, and cannot be directly used to build an index equivalent to *MQ*, unless imposing over them some arbitrary thresholds. For this reason, we preferred to just rely on the *CCBC* that, as shown in a previous study [71], provides a good indication of software modularity, sometimes even better than structural measures.

In our study, we tried to verify whether developers teams that split and merge between two releases re-organize the work in more "semantically" related files. Thus, we compute the the values of conceptual coupling for an emerging team by considering the set of files modified by the team members, and then we observe how it changes when teams split and merge.

3.2.3 Analysis Method

To address $\mathbf{RQ}_1$, given two releases r_k and r_{k+1}, we report the number and percentage of (i) teams in r_k that survive in r_{k+1}, (ii) teams in r_k that split in r_{k+1}, and (iii) mergers occurring

Table 3.2: Evolution of teams across software releases.

Releases r_k	**Teams** r_k	**Teams** r_{k+1}	**Teams disapp. in** r_{k+1}	**Teams split from** r_k **to** r_{k+1}	**Teams merged** r_{k+1}	**Teams surv. in** r_{k+1}	**New teams** r_{k+1}
Apache 2.0 → 2.2.0	9	10	5 (56%)	1 (11%)	1 (10%)	4 (44%)	2 (20%)
Apache 2.2.0 → 2.2.4	10	9	3 (30%)	2 (20%)	2 (22%)	5 (50%)	1 (11%)
Apache 2.2.4 → 2.2.12	9	8	4 (44%)	2 (22%)	2 (25%)	5 (56%)	1 (20%)
Apache 2.2.12 → 2.4.1	8	8	4 (50%)	0 (0%)	0 (0%)	4 (50%)	0 (0%)
Average	-	-	**45%**	**13%**	**18%**	**50%**	**13%**
Eclipse JDT 3.0 → 3.2	7	7	2 (27%)	1 (14%)	2 (29%)	5 (71%)	1 (14%)
Eclipse JDT 3.2 → 3.4	7	7	3 (43%)	1 (14%)	0 (0%)	4 (57%)	1 (14%)
Eclipse JDT 3.4 → 3.6	7	6	4 (57%)	1 (14%)	1 (17%)	3 (43%)	1 (17%)
Eclipse JDT 3.6 → 4.2	6	6	3 (50%)	1 (17%)	1 (17%)	3 (50%)	1 (17%)
Average	-	-	**45%**	**15%**	**16%**	**55%**	**16%**
Netbeans 2.3 → 3.0.20	11	7	6 (54%)	3 (27%)	5 (83%)	5 (46%)	0 (0%)
Netbeans 3.0.20 → 3.0.25	7	11	1 (14%)	4 (57%)	5 (45%)	6 (86%)	2 (18%)
Netbeans 3.0.25 → 3.5.0	11	10	2 (18%)	4 (36%)	2 (50%)	6 (55%)	0 (0%)
Netbeans 3.5.0 → 4.0	10	10	6 (60%)	2 (20%)	0 (0%)	4 (60%)	2 (20%)
Average	-	-	**37%**	**35%**	**45%**	**62%**	**10%**
Samba 2.3 → 3.0.20	9	9	2 (22%)	5 (56%)	3 (33%)	7 (78%)	1 (11%)
Samba 3.0.20 → 3.0.25	9	10	3 (33%)	1 (11%)	1 (10%)	7 (78%)	1 (10%)
Samba 3.0.25 → 3.5.0	10	9	2 (20%)	3 (30%)	4 (44%)	6 (60%)	0 (0%)
Samba 3.5.0 → 4.0	9	9	1 (11%)	3 (33%)	6 (67%)	6 (67%)	0 (0%)
Average	-	-	**22%**	**35%**	**35%**	**71%**	**5%**

in r_{k+1} from (sub) teams of r_k. Note that the number of mergers also considers sub-groups derived from the splits in r_k. That is, two groups t_1, t_2 of r_k can split in $t_{1'}$, $t_{1''}$, $t_{2'}$, and $t_{2''}$ (two splits occur). Then, in r_{k+1}, $t_{1'}$ and $t_{2'}$ as well as $t_{1''}$ and $t_{2''}$ merge (two mergers occur).

To address $\mathbf{RQ}_2$, we compute the average *MQ* and *CCBC* for files modified by each team before and after splits and mergers, and compare it by means of boxplots, and appropriate statistical tests, i.e., Wilcoxon rank sum paired test [81]. Specifically, we test the null hypothesis H_{0a}*: there is no significant difference between the average MQ (CCBC) before and after split*, and the null hypothesis H_{0b}*: there is no significant difference between the average MQ (CCBC) before and after mergers*. Then, we estimate the magnitude of the variation using Cliff's Delta (d) [82], a non-parametric effect size measure for ordinal data, which indicates the magnitude of the effect of the main treatment on the dependent variables. The effect size ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [83].

3.3 Analysis of the Results

This section describes the results achieved aiming at providing answers to the research questions formulated in Section 3.2.

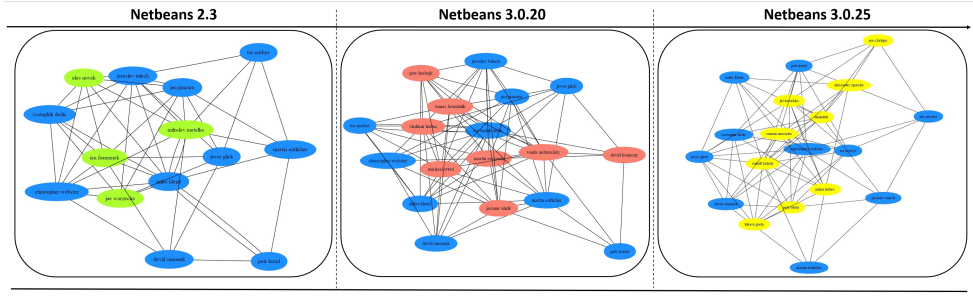


Figure 3.3: A stable groups of developers (in blue) that joined different teams during software evolution.

3.3.1 RQ$_1$: How do emerging collaborations change across software releases?

Table 3.2 reports—for each pair of releases r_k and r_{k+1} considered in our study—the number of teams identified in the discussions following the two releases, as well as the number of teams that split in release r_k, merged in release r_{k+1}, that survived in r_{k+1}, totally disappeared, and the new teams that emerged in r_{k+1}. Details about developers' involvement in the various releases are shown in Table 3.3 which reports—for each pair of releases—the number of developers that (i) were inactive during that interval, i.e., did not participate to the discussion nor they made any code change, (ii) joined the project, i.e., during that period they participated to the discussion and performed code changes, or, (iii) likely left the project, i.e., from that period until the end of our observation they never participated in the discussion nor made changes again.

The analysis of results reveals that, even if the number of teams across releases is quite stable, there is a continuous reorganization of teams during the evolution of a project. In all four projects, at least 50% of the teams, on average, survived from release r_k to r_{k+1} (for Samba and Netbeans this value is above 62%). This means that in the new release at least half of the developers tend to work in the same teams, while the remaining ones re-organize their work.

There is also a substantial number of teams that disappeared from release r_k to release r_{k+1}. This phenomenon is particularly evident for Eclipse JDT and Apache. Specifically, for Apache and Eclipse JDT the percentage of teams identified in r_k that are not present anymore in release r_{k+1} is around 50%. This means that several teams completely crumble from one release to another, i.e., some developers join different teams or leave (definitively or temporarily) the project (see Table 3.3). It can be noticed that, for Apache and Eclipse JDT, the high number of teams that disappeared is balanced by a comparable number of survived

Table 3.3: Inactive (IN), new (N), and developers that likely left the project (DL).

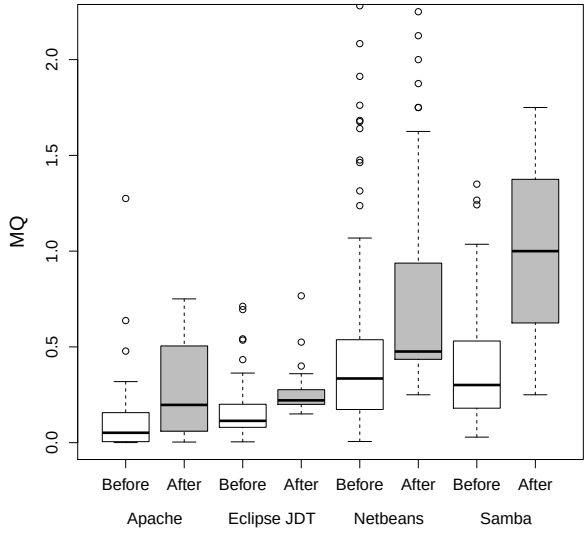
$r_k \rightarrow r_{k+1}$	IN	N	DL
Apache 2.0 → 2.2.0	2	23	2
Apache 2.2.0 → 2.2.4	4	10	3
Apache 2.2.4 → 2.2.12	15	5	6
Apache 2.2.12 → 2.4.1	2	8	6
Eclipse JDT 3.0 → 3.2	2	7	1
Eclipse JDT 3.2 → 3.4	3	2	2
Eclipse JDT 3.4 → 3.6	6	3	4
Eclipse JDT 3.6 → 4.2	1	2	1
Netbeans 2.3 → 3.0.20	5	23	7
Netbeans 3.0.20 → 3.0.25	3	41	6
Netbeans 3.0.25 → 3.5.0	13	45	2
Netbeans 3.5.0 → 4.0	5	10	4
Samba 2.3 → 3.0.20	4	27	13
Samba 3.0.20 → 3.0.25	5	17	12
Samba 3.0.25 →3.5.0	2	35	4
Samba 3.5.0 → 4.0	15	8	24

Table 3.4: Change of MQ and CCBC when teams split: Wilcoxon test results and Cliff's d.

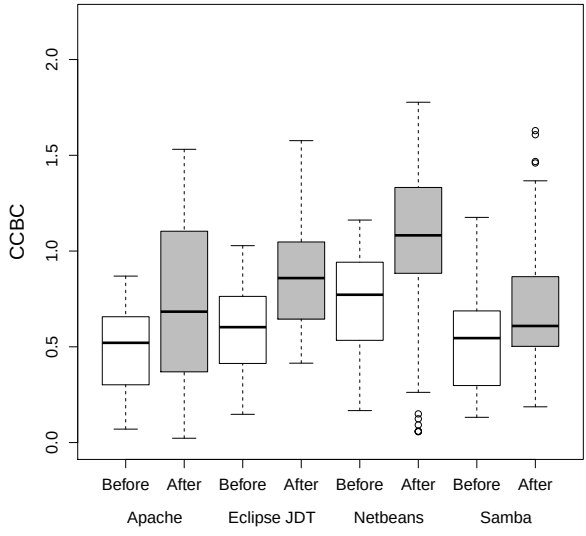
Project	**MQ**		**CCBC**	
	p-value	**Cliff's d**	**p-value**	**Cliff's d**
Apache httpd	<**0.01**	57 (large)	<**0.01**	32 (small)
Eclispe JDT	<**0.01**	64 (large)	<**0.01**	0.5 (large)
Netbeans	<**0.01**	59 (large)	<**0.01**	0.67 (large)
Samba	<**0.01**	25 (small)	<**0.01**	0.40 (medium)

teams (with few new emerging teams). Specifically, for such projects about 50% of the teams in release r_{k+1} is completely new, i.e., composed of developers belonging to different teams or of newcomers, as can be noticed in Table 3.2. However, if we consider all the projects in this study, the number of teams disappeared in a project is very often much lower of the number of survived teams. For all the projects, the number of teams that disappear in a version r_k and the number of teams that emerge in r_{k+1} is much lower, suggesting that a part of the existing teams tend to recombine.

To provide a qualitative evidence of the above observations, we inspected the source code change history and mails for the above identified teams. Figure 3.3 shows a group of nine Netbeans developers (highlighted in blue) that joined different teams in three subsequent releases (2.3, 3.0.20, and 3.0.25). As we can notice such developers always worked together, albeit moving across different teams, i.e., involving other people in different releases.

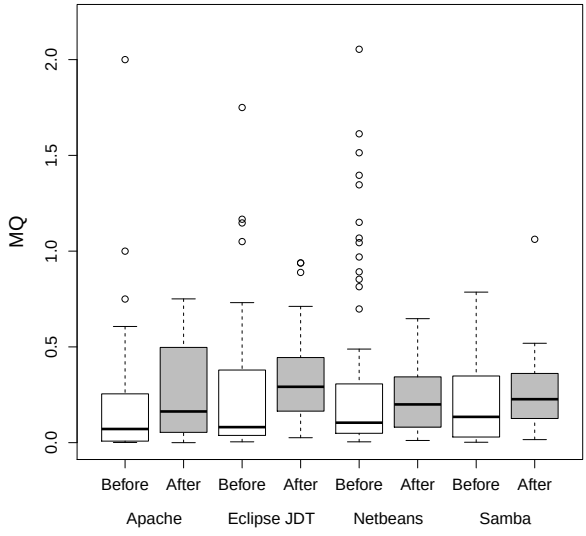


(a) MQ

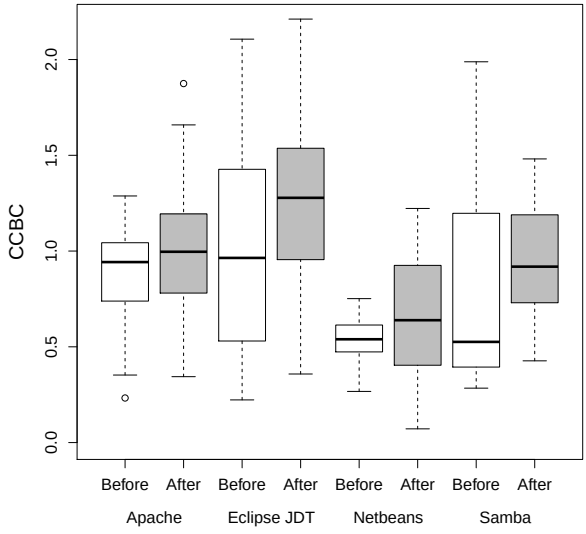


(b) CCBC

Figure 3.4: MQ and CCBC before and after team splits.



(a) MQ



(b) CCBC

Figure 3.5: MQ and CCBC before and after team mergers.

Table 3.5: Change of MQ and CCBC when teams merge: Wilcoxon test results and Cliff's d.

Project	MQ		CCBC	
	p-value	Cliff's d	p-value	Cliff's d
Apache httpd	**0.036**	0.40 (medium)	**0.02**	0.29 (small)
Eclipse JDT	0.07	0.32 (small)	**0.03**	0.18 (small)
Netbeans	0.11	0.45 (medium)	**0.01**	35 (medium)
Samba	0.09	0.04 (small)	**0.01**	40 (medium)

RQ_1 summary: *Teams continuously evolve during the evolution of a software system. We observed that groups of people build stable working relations (at least 50% of the teams survive between two subsequent releases), and occasionally work with others*).

3.3.2 RQ_2: How does the evolution of emerging collaboration relate to the cohesiveness of files changed by emerging teams?

Figure 3.4 shows boxplots of how *MQ* and *CCBC* vary after teams split. Table 3.4 complements Figure 3.4 with results of Wilcoxon test and Cliff's d effect size values. Results for all four projects indicate that when teams splits, *MQ* and *CCBC* significantly increase with a large effect size. Therefore for both metrics we can reject the null hypothesis H_{01}.

For the mergers, Figure 3.5 shows boxplots of *MQ* and *CCBC* variations, while Table 3.5 reports results of Wilcoxon test and Cliff's d effect size values. In this case, we did not observe a significant increase of *MQ* (except for Apache). Hence, it is not possible to reject H_{0b} for what concerns *MQ*. In the case of team merger one would expect that the *MQ* could even decrease, because the focus of the new team will likely be broader. However, not only this does not happen, but also *MQ* slightly increases. Also, we can observe how the *CCBC* always significantly increases (hence we can reject H_{0b} in this case), although the effect size is medium for Netbeans and Samba, and small in the other cases. A possible interpretation to the results obtained for the mergers is that the previous teams join their force to work on a new, focused task.

To provide qualitative explanations to the above results, we inspected source code changes and mails for the above identified teams. Figure 3.6 depicts a scenario where two Samba teams merge between release 2.3 and 3.0.20. The analysis of the file changed by each team revealed that:

- *Team 1* is a testing team, i.e., a team of developers that is called very often to test some new components or features. The developers of this team, between the various

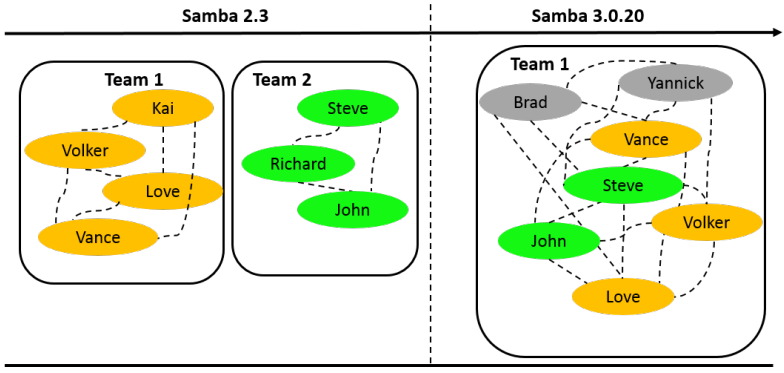


Figure 3.6: Example of team merger in Samba: as Team 1 no longer need to develop test cases (mostly available in release 3.0.20).

commitments, mainly worked on the *testsuite* folder of the Samba project. Specifically, they wrote test suites for the modules *libsmbclient*, *smbd*, and *nsswitch*.

- *Team 2* is a development team. The analysis of the change logs highlights that the development effort of this team is mainly focused on the modules *auth*, *nmbd*, *nsswitch*, *smbd* and *sam*.

This suggests that in Samba 2.3 there is an evident effort in the development of test cases, that required a group of developers to work mainly on testing activities.

In addition, from the analysis of test cases developed by *Team 1*, we can derive a “latent" relationship between *Team 1* and *Team 2*, i.e., the testing team provided test cases for some modules developed by *Team 2*.

In release 3.0.20, *Team 1* and *Team 2* merged. As a consequence of such a merger, the values of both *MQ* and *CCBC* increased. Specifically, the *MQ* increased of 73%, from an average of 0.71 to 1.23. The *CCBC* increased of 53%, from an average of 0.43 to 0.66. We expect that this variation depends on the fact that developers of the two past teams in the new release focus their effort in files that are structurally and semantically more related. The analysis of changes performed by the new team indicated less activity on the development of test suites, likely because such test suites were mostly produced in the previous release, hence they only needed to be repaired where necessary. In addition, the files changed by the new emerging team in release 3.0.20 revealed that the developers focused the development activity on the module `heimdal`. In release 3 Samba used the external Kerberos authentication. Starting from such a release the developers worked on the integration with Heimdal (an implementation of Kerberos 5), integration that was fully available only in release 4. In this

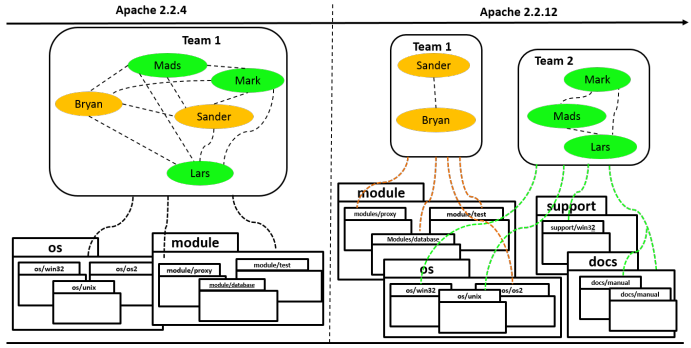


Figure 3.7: Example of team split in Apache httpd from release 2.2.4 to release 2.2.12.

case the reorganization of teams is triggered by the need to deal with changes in the project design and implementation.

We found cases in which, when teams split, one of the metrics (*CCBC*) significantly increased, while the other (*MQ*) did not, or vice versa. Specifically, let us to consider the Apache httpd project, where there are few cases in which emerging teams that are obtained by a split of a previous emerging team (from the previous release) had an increase of the resulting *CCBC* with a corresponding decrease of *MQ*. For example, as it can be seen from Figure 3.7, from release 2.2.4 to release 2.2.12, a team of five developers (Bryan, Lars, Mads, Mark, Sander) is split into two smaller teams of two (Bryan and Sanders) and three developers (Lars, Mads and Mark) respectively. The team of release 2.2.4 is mainly focused on some sub-modules of `os` (e.g., `os/win2`, `os/unix`, `os/os2`) and `os` (e.g., `modules/proxy`, `modules/test`, `modules/database`). In the new release, the team splits into two sub-teams. Both teams continue to work on sub-modules of `os`, however separating their activities, i.e., the first team works on other modules of `modules/` (e.g., `modules/proxy`, `modules/test`, `modules/database`) and the second on `support/win32`, `docs/manual and docs/conf` modules. At the same time, both teams also work on some `os` modules. It happens that the modules modified by each of these two teams are somewhat decoupled (therefore, the *MQ* decreases by 45%, from an initial *MQ* of 0.29 to an average *MQ* of 0.20 between the two teams). However, the conceptual cohesion is high, in that such modules share several concepts. For example, we found an increase of shared common words between the `os`, `support`, `docs` modules, terms mainly related to code changes for support windows platform (*winapi*, *windowsStation* and so on). As a consequence, the *CCBC* value increases of more than times its value (from an initial *CCBC* of 0.36 to a *CCBC* of 0.86).

RQ_2 summary: *Both MQ and CBCC increase when teams split, while MQ does not significantly changes and CCBC increases when teams merge. This means that team reorganization relates to work on more (structurally and semantically) cohesive sets of files.*

3.4 Threats to Validity

This section discusses the threats to validity of the study reported in this study, categorized into threats to *construct*, *conclusion*, *internal*, and *external* validity.

Threats to *construct validity* concern the relationship between theory and observation. Previous work in this area used either mailing lists [68] or issue trackers [36] to identify the developers' communication network. To mitigate the risk of missing links, in this study we consider both sources of information. Although communication in open source projects is usually asynchronous [26] and therefore the aforementioned communication channels should be considered as representative enough of the actual developers' communication, we are aware that written communication through mailing lists can constitute a partial view of the overall communication, as also reported in a study by Aranda and Venolia [34]. However, such a problem is more crucial in industrial projects than in OSS projects where emails constitute the premier mean of communication for developers working around the world.

As explained in Section 4.2.2, we mitigated imprecision and incompleteness of the mapping and unification of mailing lists and issue trackers contributor names, and of their mapping onto author IDs available in the versioning system, by performing a manual validation. Besides that, it is important to note that we considered as authors only those mentioned in the versioning system. However, we cannot exclude that code changes were authored by somebody not mentioned there.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Where appropriate—i.e., for $\mathbf{RQ}_2$—we use appropriate (non parametric) statistical tests and effect size measures to support our findings. In other cases, we analyze the phenomenon by using descriptive statistics.

Threats to *internal validity* concern factors that can affect the results. The study is purely observational and we cannot claim a causation between the observed phenomenon, e.g., team splits/mergers and increase of *MQ* and *CCBC* as observed in $\mathbf{RQ}_2$. However, we qualitatively support such finding by some manual analysis of source code changes and of emails exchanged between developers.

Threats to *external validity* concern the generalization of findings. In Section 3.2 we have motivated the choice of the four open source projects considered in this study. However, the

obtained findings have a validity confined to considered OSS projects, and above all cannot be directly extended to some commercial projects where the face-to-face communication is more than often used in place of written communication. Having said that, many companies are increasingly using written communication in issue trackers or mailing lists as a way to record the decisions taken in a project, hence in such situations what described in this study is still applicable.

3.5 Related Work

An open source community can be considered as a complex, self-organizing system. Specifically, such systems are typically comprised of large numbers of locally interacting elements, where developers the main components in this network [84]. This has motivated a lot of research effort in mining the social networking among developers (a more detailed related work about analysis of DSN is reported in Section 2.5).

Capra and Wassermann [26] characterize—by means of a survey involving 70 projects—management activities in both commercial and open source organizations. They analyze mechanisms for communication and collaboration, as the internal organizational structure of the projects. In the investigated open source projects, communication occurs through asynchronous tools such mailing lists, although IRC channels are also used, and physical meetings are sometimes organized. Also, they found that the organizational structure of some open source projects is not much different from those of some closed source projects.

Crowston and Howison [61] use co-occurrence of developers on bug reports as indicators of a social link. They observe that development teams vary widely in their communications centralization, from projects completely centered on one developer (usually smaller projects) to projects that are highly decentralized (usually larger projects). Such a finding inspired the study by Bird *et al.* [68] that analyze the relationship between communications structure and code modularity. Their results show that the sub communities identified using communication information are significantly connected with collaboration behavior. In our study, we analyze how the connectedness between communication structure and code cohesiveness evolve, although as explained in Section 3.2 we use different measures (*MQ* and *CCBC*) than what used by Bird *et al.* (activity focus).

Hong *et al.* [36] compare developers' network events with events occurring in traditional social networks such as Facebook or Twitter. Our study shares with Hong *et al.* [36] the analysis of how emerging teams of developers re-organize themselves over time. However, we specifically related such a reorganization with the cohesiveness of files they modify, measured in terms of *MQ* and *CCBC*.

Singh [50] analyzes over 4,000 projects from SourceForge to understand how the relations among developers influence development activities. His results show that "small-world communities"—i.e., projects where there are small clusters of developers discussing with each other—are a factor for the project success. We share with this work the importance of developers' discussion during development and the fact that such a discussion, possibly, occurs in small clusters related to specific components or to specific issues to be handled.

Surian *et al.* [85] mine collaboration patterns from a large developers' network in SourceForge. They find that not all developers are connected to every other developers and that there are many collaboration clusters. Their findings are consistent with those of Singh *et al.* [50]; that is, the small-world phenomenon also exists in SourceForge, especially when developers in a network are separated, on average, by approximately 6 hops.

The methodology used in this study to analyze the evolution of teams might complement all these approaches. In particular, the analysis of the communication network can be exploited not only to identify roles in developer teams but also to identify how teams evolve during software evolution. The evolution of the teams can be analyzed by a project manager in order to monitor the "project landscape" [25] and facilitate the integration of newcomers.

3.6 Summary

Differently from commercial projects, where teams are often designed by project managers, open source projects are characterized by spontaneous collaborations, that result in emerging teams, often reflected by frequent communication through asynchronous channels such as e-mails and communications in issue trackers.

This study investigated, by analyzing software repositories of four open source projects—namely Apache httpd, Eclipse JDT, Netbeans and Samba—how such emerging teams evolve over time, and to what extent such evolution relates with the developers' activity on source code. Results of the study indicate that:

- Emerging teams tend to recombine over time. We often observed groups (e.g., of two or more developers) having a stable collaboration over time, although involving each time different other people, i.e., moving between different emerging teams.

- Team splits correspond to a significantly higher cohesiveness of the files teams modify; specifically, we observed a significant increase of both Modularization Quality (MQ) and conceptual coupling between classes. The qualitative analysis indicated that the new emerging teams formed after a split work on closer files than the original ones.

- Team mergers do not imply a decrease of structural and conceptual cohesiveness. On the contrary, we found that in general *MQ* does not change, or it slightly improves, while the *CCBC* still increases. Qualitative analysis suggested that teams merged when they were working on related files and specific circumstances lead developers of one or both teams to change their activities. This is for example the case when developers having worked in a period over test cases join their forces with those working on production code as there is less need to work on tests. In other cases, team reorganization can be dictated by changes in the system design and/or implementation.

- in general the evolution/reorganization of emerging teams of developers (identified by analyzing communication channels) over the time is reflected in cohesive changes in the source codes. This result, motivates our conjecture about the possibility to do refactoring/re-modularization of source code by analyzing social interaction between developers. Specifically, in our previous work [15] we present a technique to suggest refactoring based on team co-maintenance patterns. If two teams work on the same module (package, class, method), then the tool can suggest that this module can be refactored based on the activity patterns. We evaluate the approach using five projects from the Android API. Differently from this work we want to refactor modules identifying teams relying on social links extracted from mailing lists and issue trackers data.

The obtained results shed the light on the collaboration dynamics in open source projects, highlighting how such dynamics can be influenced by various factors, that could be the attitude of small teams of developers to work together, or changes in specific project needs, e.g., performing testing vs. development activities.

Chapter 4

How Developer Communications are Used to Support Third-Party Libraries

Contents

In the Chapter 3 we empirically investigated how, when a project evolves, emerging teams re-organize themselves—e.g., by splitting or merging. We relate the evolution of teams to the files they change, to investigate whether teams split to work on cohesive groups of files. Results of this study—conducted on the evolution history of four open source projects suggest that emerging team mergers and splits imply working on more cohesive groups of files. Such indications serve to better understand the evolution of a software project by project newcomer. More important, the observation of how emerging teams change can serve to suggest software remodularization/re-factoring actions for newcomers/senior developers that are interested to better restructure the software components.

In this Chapter we study the evolution of an OSS projects ecosystem of the Apache Software Foundation, in terms of number of developers, their interactions and the dependencies between projects. Our goal is to understand how developer communications are used to support TPL in such projects ecosystem. Specifically, we analyze a subset of the Apache ecosystem, consisting of 147 projects, for a period of 14 years, resulting in 1,964 releases. Specifically, we investigate (i) how dependencies between projects and the developers interactions evolve over time when the ecosystem grows, (ii) what is the relation between sub-projects developers overlap and sub-projects dependencies, (iii) how developers discuss the needs and risks of such dependencies upgrades.

The study results—qualitatively confirmed by observations made by analyzing the developers' interactions and discussions— suggest the a proper communication between developers belonging to different sub-projects of the ecosystem is one o the key element that avoid the presence of bugs and/or fault, as well as, incompatibility problem between projects of the same ecosystem. Such information can be used for define an approach that help developers/newcomers to avoid changes that could break the dependency with TPL. Further analysis are performed to investigate what are the product and process factors that can likely trigger dependency upgrades [2, 9]. However, we do not report such results in this chapter because are out of the focus of this thesis.

4.1 Motivation: analysis of developers collaborations and its impact/relation on projects dependencies

Software development is a collaboration-based activity. The highest grade of such a collaboration can perhaps be achieved when a software company decides to make available their product line architecture and shared components to external parties. Making available their own product outside the organizational boundary generates the so-called software ecosystems [86, 87]. In other words, a software ecosystem is a group of software projects that are developed and co-evolve in the same environment. These projects share source code, depend on one another, and can be built on similar technologies. In some cases, they generally have a closed core that provides the basic functionality, and a set of components that provide specific functionality. For example, the Eclipse project provides the core functionality of an IDE, that can be customized into any kind of IDE or editor though a specific plug-in. In other cases, they can be completely different projects sharing a set of common components.

Software ecosystems are therefore a new dimension of collaboration, that allows companies to satisfy the need of their customers as rapidly as possible and facilitate mass customization [86]. Thus, in recent years it is possible to observe an increasing trend of software companies that are moving from product lines towards software ecosystems to better support the intra-organizational reuse of software. Such a transition recalls the need of methods and tools to effectively manage both the coordination and the evolution of software ecosystems.

A crucial activity for an effective evolution of a software ecosystem is managing the upgrades of libraries/components. When one project undergoes changes and issues a new release, this may or may not lead other projects to upgrade their dependencies. On the one hand, using up-to-date releases of libraries/components may result useful, because these releases can contain new and useful features, and/or possibly some faults may have been fixed. On the other hand, the upgrade of a component may create a series of issues. For example, some APIs may have changed their interface, or might even be deprecated [88], which requires the adaptation of its client.

In addition, let us suppose a program uses multiple libraries, namely lib_1 and lib_2, and lib_1 depends on lib_2. It can happen that if one upgrades lib_2, then lib_1 no longer works because does not support the new release of lib_2. Last, but not least, a library/component might have changed its license making it legally incompatible with the program using it [89]. All these scenarios suggest that updating a library/component in large ecosystems is a complex and daunting task, which requires to ponder several factors.

Following the recent trend of studies aimed at analyzing the evolution of software ecosystems [88,90,91], we present an exploratory study conducted on the Java subset of the Apache

ecosystem focusing the attention on how and why dependencies (i.e., dependencies related to API usage and/or framework usage through extension) between software projects evolve. The entire Apache ecosystem is composed of 195 software projects developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases.

In our study we analyzed how the number of projects, their size, the dependencies among them, and the number of active developers changed in the ecosystem during time. After this preliminary analysis, we study also the relation between sub-projects developers overlap and sub-projects dependencies. We analyzed mailing lists and issue tracking systems in order to understand to what extent developers discuss the management of dependencies and what are the factors subject of the discussion. Such an investigation required the manual analysis of 7,685 discussions and allowed us to provide some qualitative insights on the evolution of dependencies in the Apache ecosystem.

The obtained results indicate that pairs of projects having a dependency share a higher number of developers as compared to pairs of projects do not having a dependency. However, such an overlap does not influence the client's upgrade frequency. These observations are also confirmed by the analysis of communications between developers.

The Chapter is organized as follows. Section 4.2 describes the study definition and planning, while results are reported in Section 4.3. Section 4.4 discusses the threats that could affect the validity of the results achieved. Section 4.5 presents the existing literature about the evolution of software ecosystems and evolution/adaptation of APIs. Finally, Section 4.6 summaries the results of this study.

4.2 Study Definition and Planning

The *goal* of this study is to analyze the evolution of developers communities in a software ecosystem, with the *purpose* of understanding how dependencies are discussed/maintained. The *quality focus* is software maintainability, which could be improved by understanding how developer discuss about software upgrades in mailing lists and issue trackers. The *perspective* is of researchers interested in understanding when and why developers upgrade dependencies in software ecosystems.

The *context* of the study consists of the entire history of the Java subset of the Apache ecosystem, that represents the vast majority of it (75% of the projects). To date, the entire Apache ecosystem is composed of 195 software projects spread over 23 different categories (e.g., big-data, FTP, mobile, library, testing, XML) and developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems,

in the period of time going from June 1999 to April 2013 resulting in 1,964 releases. The size of the ecosystem in the analyzed period of time ranges from 32 up to 28,584 KLOCs, while the number of classes (methods) ranges from 113 to 114,000 (1,386 to 780,731). For sake of clarity, in the following we refer to the project having a dependency toward another project as the "client project" and to the project used by a "client project" as the "library project".

4.2.1 Research Questions

The study aims at providing answers for the following three research questions:

- **RQ**$_1$: *How does the Apache ecosystem evolve?* This research question is preliminary to the other three, and aims at providing a context for our study. Specifically, we analyze how the number of projects, their size, the dependencies among them, and the number of active developers changed in the Apache ecosystem during time. Such information represents the foundation for the other research questions.

- **RQ**$_2$: *What is the relation between sub-projects developers overlap and sub-projects dependencies?* Our conjecture is that pairs of projects having a dependency share a higher number of developers as compared to pairs of projects do not having a dependency

- **RQ**$_3$: *How are dependencies discussed between developers?* This research question aims at understanding to what extent is the management of dependencies between a client and a library discussed by developers over mailing lists and issue trackers, analyzing the factors object of the discussion and the developers involved.

4.2.2 Data Extraction and Analysis

To answer our research questions we first *download the source code* of the 1,964 software releases considered in our study. We use a crawler and a code analyzer developed in the context of the MARKOS European project[1] [92]. The crawler is able to identify for a given project of interest the list of available releases with their release date as well as its SVN address. This information is extracted by crawling DOAP (Description Of A Project) files available on the Internet[2].

Using the information extracted by the crawler, the code analyzer checks-out files from the SVN repository and identifies the folder containing each of the project releases identified

[1] `http://www.markosproject.eu`
[2] `http://projects.apache.org/doap.html`

by the crawler. This is done by exploiting the SVN tag mechanism. In other words, the versioning system of Apache projects has a separate directory for each release (where files belonging to such a release are stored), besides keeping the project history in the SVN main trunk. In case the code analyzer does not identify any folder containing a particular release, it reports the problem. This issue occurred for 278 releases (across all projects) that were manually downloaded from the Apache release archives, available online for each project[3].

Once downloaded all the software releases, we *extract dependencies* existing between such releases. Note that in this study we focus on dependencies existing between Java Apache projects, ignoring those toward projects external to the Apache ecosystem or not written in Java. Also in this case, the MARKOS code analyzer has been used. The identification of the inter-project dependencies is performed in different steps. Given a set of software releases, the code analyzer searches—in each folder release—for files that explicitly report inter-project dependencies. These files in the Apache ecosystem are generally of three types: `libraries.properties`, `deps.properties`, or the Maven `pom.xml` files. Note that the dependency information reported in these files is generally detailed (i.e., both the name of the project as well as the used release are reported) and reliable.

When the code analyzer is not able to find any of these files, it searches for all jar files contained in the release folder and attempts at matching each of those files with one of the other software releases provided. This is done by computing the Levenshtein distance [93] between the name of the jar archive and the name of each provided release. The output of the code analyzer is a list of candidate dependencies between the set of provided software releases.

In our study, we assume that the dependencies extracted by parsing the files `libraries.properties`, `deps.properties`, and `pom.xml` are correct. Instead, when the dependencies are extracted by analyzing jar files in the release folder, we manually validate and classify them as *true dependencies* or as *false positives*. This operation was done by two of the authors that analyzed a total of 3,742 dependencies, classifying 832 correct dependencies. Overall, the final number of dependencies found in the analyzed 14 years of observation and considered in our study is 3,514 (i.e., 2,682 extracted from dependencies files, plus the 832 manually verified).

After having performed this first data analysis (necessary for all research questions), we perform analyses specific to each research question, explained in the following.

[3]An example of archive for the Ant project can be found here `http://archive.apache.org/dist/ant/source`

$\mathbf{RQ}_1$ analyses

To answer $\mathbf{RQ}_1$ we analyze the history of the Apache ecosystem, considering snapshots captured every month. In particular, starting from June 1999, we compute, with a granularity of one month (which we consider sufficient to observe the evolution of the ecosystem over several years):

[1] the number of existing projects;

[2] the size of the ecosystem in terms of KLOCs;

[3] the dependencies existing between projects.

To analyze how the number of developers changed during time ($\mathbf{RQ}_1$), we extract the list of *active developers* that worked in the ecosystem during its entire history[4]. In particular, from our starting date (i.e., June 1999), we compute the number of active developers at time intervals of six months (e.g., from June 1999 to January 2000). We consider a developer active in the time interval of interest if she performed at least one commit in one of the Apache projects existing at date. Note that, while we consider a granularity of one month for most of the measures, we check the activity of developers for a six months period, because the lack of activity for a short period (i.e., one month) can just occur by chance. Note also that checking whether a developer is active in a given time period does not mean determining whether a developer has left a project or not. In other words, a developer may not be active in a given time period, but she can still be part of the project and likely contribute in the future.

$\mathbf{RQ}_2$ analyses

Some of the data needed to answer $\mathbf{RQ}_2$ (e.g., dependencies, developers working on the various projects) is already available after having performed the general data extraction and the $\mathbf{RQ}_1$-related data extraction as described above. Specifically, we describe how we measure the dependent and independent variables that concern the analyses of $\mathbf{RQ}_2$, and explain the kinds of statistical analyses we perform.

Dependent Variable: Upgrades. Given the dependencies existing between different project releases, we distinguish releases of the libraries that are upgraded by client projects (hereby referred as *upgraded releases*) and releases ignored by client projects (hereby referred as *not upgraded releases*).

[4]Note that we limit our analysis to developers we can detect through their activities in the versioning system, as also pointed out in Section 2.4.

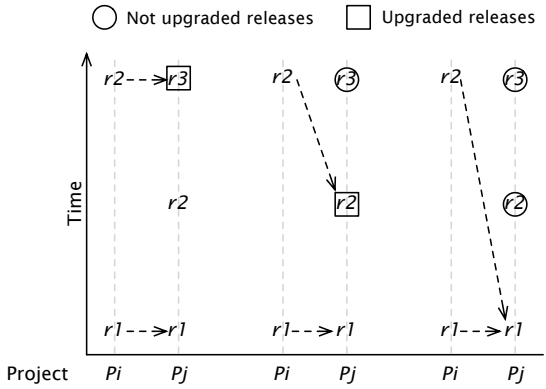


Figure 4.1: Process used to divide *upgraded* and *not upgraded* releases.

To create the two sets of releases (i.e., *upgraded releases* and *not upgraded releases*) we adopt the process depicted in Fig. 4.1. For each pair of Apache projects, P_i and P_j, having at least one dependency between their releases, when P_i upgrades the dependency towards P_j, we determine whether P_i upgrades the dependency toward to the last existing release of P_j or to another release. In the former case, we put the upgraded P_j release in the set *upgraded releases*. Instead, when the upgrade was not toward the last available release we still put the upgraded P_j release in the set *upgraded releases*, however we also put the newer ignored releases of P_j in *not upgraded releases*.

To better understand how we compute such sets, Fig. 4.1 shows three different evolution scenarios of dependencies between two projects P_i and P_j. Let us assume that the release $r1$ of P_i depends on the release $r1$ of P_j. Then, a new version of project P_i is released ($r2$). In the first scenario, when $r2$ for P_i is released, its dependency is upgraded to $r3$ of P_j, the last available P_j release. In this case, $r3$ is included in the set *upgraded releases*, while no releases are added to the set *not upgraded releases*, since P_i correctly upgraded its dependency to the last available P_j release. In the second scenario (reported in the middle of Fig. 4.1), the release $r2$ of P_i upgrades its dependency to the release $r2$ of P_j, even if a newer release (i.e., $r3$) is available. In this case the release $r3$ of P_j has been "ignored" by P_i and thus, it is added to the set *not upgraded releases*, while release $r2$ of P_j is added to the set *upgraded releases*. In the third and last case, P_i does not upgrade at all the dependencies toward P_j, i.e., the new release of P_i continues to use the release $r1$ of P_j, despite the availability of more recent releases (i.e., $r2$ and $r3$). In this case, $r2$ and $r3$ are added to the set *not upgraded releases*, while no releases are added to the set *upgraded releases*.

Note that, if a release r_i of a project P_j belongs to the set of *upgraded releases* when

analyzing dependencies between P_i and P_j, and the same release belongs to the set of *not upgraded releases* when analyzing dependencies between a project P_s and P_j, the release r_i is removed from both sets, and not considered any longer in the comparison between *upgraded releases* and *not upgraded releases*. This is done (i) to avoid overlap between the two sets (which would make the comparison unfair); and (ii) to strongly isolate only releases that are generally upgraded (and not) by client projects.

Independent Variable - Developers' overlap. We analyze the overlap in terms of active developers (already detected to answer $\mathbf{RQ}_1$) between all pairs of projects existing in the Apache ecosystem. Given two projects C and L, the developers' overlap (in percentage) between them is computed as:

$$overlap_{C,L} : \frac{|D_C \cap D_L|}{|D_C \cup D_L|}$$

where T_C are the developers of project C and T_L are the developers of project L. With this analysis we want to understand if (i) pairs of projects *having* a dependency share more/less developers than pairs of projects *do not having* a dependency and (ii) client projects having a high overlap of developers with the libraries they use have a higher upgrade frequency (still by using the Spearman correlation).

$\mathbf{RQ}_3$ analyses

Concerning $\mathbf{RQ}_3$, we downloaded [5] the Apache mailing lists and the discussions on the Apache issue trackers for the projects object of our study showing at least a dependency. This resulted in the download of 84 mailing lists and nine issue trackers. Note that the number of issue trackers downloaded is considerably lower than the number of mailing lists, due to the fact that most of the Apache projects use Jira[6] as issue tracker, which automatically forwards discussions to the projects' mailing list. Therefore, in such cases it was sufficient to limit our analyses to mailing list only. Instead, this is not the case for projects using Bugzilla[7]. Overall, we downloaded 664,490 discussions containing a total of 1,924,002 messages.

Then, for each pair of projects C, L exhibiting a dependency, we filter—from the project C mailing lists and issue tracker—all discussions containing (in the mail object/issue title or in the mail body/issue description) the name of project L. Similarly, we filter—from the project L mailing lists and issue tracker—all discussions containing (in the mail object/issue

[5] `http://mail-archives.apache.org/mod_mbox/`
[6] `https://issues.apache.org/jira/secure/Dashboard.jspa`
[7] `http://www.bugzilla.org/`

Table 4.1: Tags assigned to classify the mailing lists discussions.

Tag	Applied when	Constraints
	GENERAL TAGS	
client	The discussion is in the client mailing list	[if !*library*]
library	The discussion is in the library mailing list	[if !*client*]
dependency	The discussion focuses on the dependency between the client and the library	-
	DEVELOPERS TAGS	
only developers client	only developers of the client project take part to the discussion	[if *dependency* && !*only developers library* && !*both developers*]
only developers library	only developers of the library project take part to the discussion	[if *dependency* && !*only developers client* && !*both developers*]
both developers	both developers of the client and of the library projects take part to the discussion	[if *dependency* && !*only developers library* && !*only developers client*]
	TOPIC TAGS	
break	the discussion is about avoiding changes that could break the dependency	[if *dependency* && !*fix* && !*upgrade* && !*use* && !*other*]
fix	the discussion is about changes needed to fix a dependency	[if *dependency* && !*break* && !*upgrade* && !*use* && !*other*]
upgrade	the discussion is focused on whether upgrading/not upgrading a dependencies toward a new available release of the library project	[if *dependency* && *client* && !*fix* && !*break* && !*use* && !*other*]
use	the discussion is about how to use the library project	[if *dependency* && *client* && !*fix* && !*upgrade* && !*break* && !*other*]
other	the discussion is about the dependency, but cannot be classified with any of the previous tags	[if *dependency* && !*break* && !*fix* && !*upgrade* && !*use*]

title or in the mail body/issue description) the name of project C. This resulted in 7,685 discussions that have been manually analyzed by two of the authors, and classified using the tags shown in Table 4.1.

The manual analysis has been performed as follows. Firstly, *general tags* are assigned to the discussion, classifying it as belonging to the *client* or to the *library* mailing list/issue tracker and, most importantly, verifying if the discussion is focused on the management of the dependency between the client and the library project (tag *dependency* in Table 4.1).

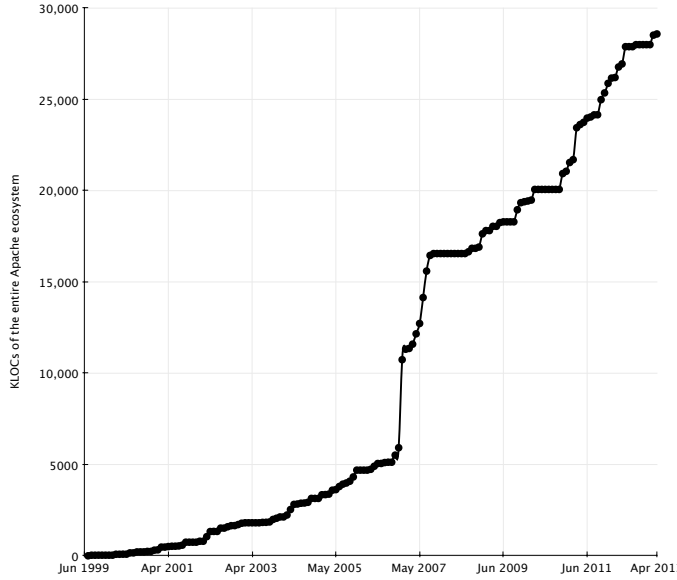


Figure 4.2: Evolution of the size in the Apache ecosystem.

Then, if the tag *dependency* has been assigned to the discussion, developers and topic tags are also associated to it. Developer tags aim at classifying the developers taking part to the discussion, while topic tags categorize the aim of the discussion (see Table 4.1). Developer tags have been automatically assigned by matching the email addresses used in the mailing lists/issue trackers with those used in the versioning systems of the project under study. We report descriptive statistics of the tags assigned to the analyzed discussions, and then discuss the most interesting cases. Note that results of the manual analysis performed in the context of this research question can also help in corroborating quantitative findings from $\mathbf{RQ}_2$.

4.3 Analysis of the Results

This section discusses the study results, in order to answer the four research questions formulated in Section 4.2.

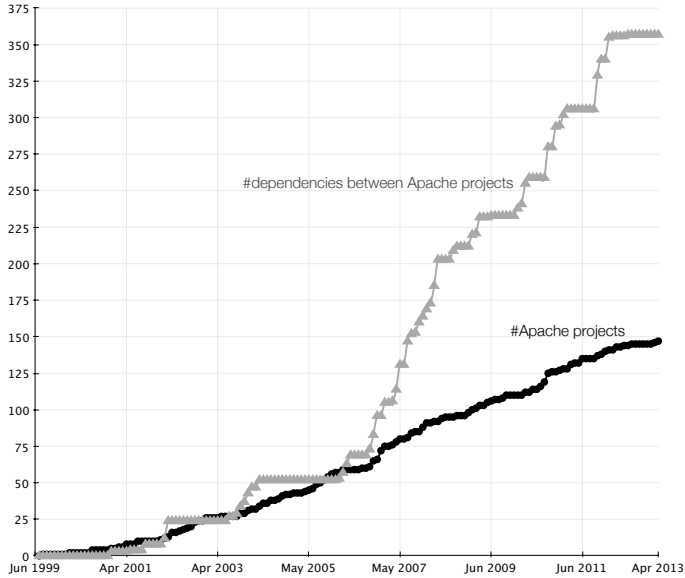


Figure 4.3: Evolution of the projects and dependencies in the Apache ecosystem.

4.3.1 RQ_1: How does the Apache ecosystem evolve?

Fig. 4.2, 4.3, and 4.4 report the evolution over time of the Java Apache ecosystem, in terms of size measured in KLOCs (see Fig. 4.2), number of projects (black line in Fig. 4.3), number of dependencies existing between them (gray line in Fig. 4.3), and number of active developers[8] (Fig. 4.4).

As expected, during the analyzed 14 years, the size of the Apache ecosystem (Fig. 4.2) grows up exponentially (model fitting resulted in an adjusted $R^2 = 0.56$). From the single Java project existing in 1999 (i.e., APACHE ECS[9]) the Apache ecosystem grows up to the 147 Java projects existing today. Such a growth is linear (adjusted $R^2 = 0.98$). With the number of projects also the size—see Fig. 4.2—of the entire ecosystem grows, by reaching almost 30 Million LOCs in April 2013. A very strong peak in the size of the ecosystem can be observed between the end of 2006 and the begin of 2007, when the Apache ecosystem doubled its size. In this period, several new, large project have been added to the ecosystem.

[8]Remember that we consider a developer "active" if she performed at least one commit in the previous six months.

[9]`http://projects.apache.org/projects/ecs.html`

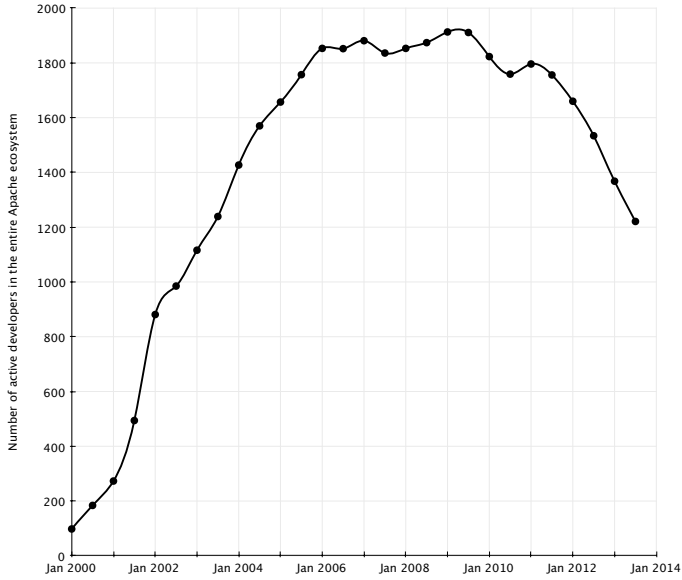


Figure 4.4: Evolution of active developers in the Apache ecosystem.

Examples include APACHE UIMA[10] with its two millions of LOCs and APACHE DERBY[11] with almost one million LOCs.

Fig. 4.4 shows that the number of active developers grows exponentially (adjusted $R^2 = 0.82$) until January 2006 together with the increase of the number of projects (Fig. 4.3) in the Apache ecosystem. In particular, in 2006 there were 58 Java projects carried out by almost 1,800 developers. Then, while the number of projects continued its linearly growth (see Fig. 4.3), the number active developers stopped its growth, and remained almost stable for four years at a level of 1,800 people. Then, from 2011 beyond we observed a decrease of the number of active developers. At the time of writing (November 2013) such a number is of around 1,200 units.

Given the continuous increase of the number of projects in the ecosystem, this result might appear counterintuitive. We found two possible interpretations for that. The first one is related to the developers' overlap existing between the Apache projects. As previously mentioned, the developers base in 2006 was very large (1,800 people), therefore when new projects were added to the ecosystem, it is likely that they were mostly carried out by de-

[10] `http://uima.apache.org/`
[11] `http://db.apache.org/derby/`

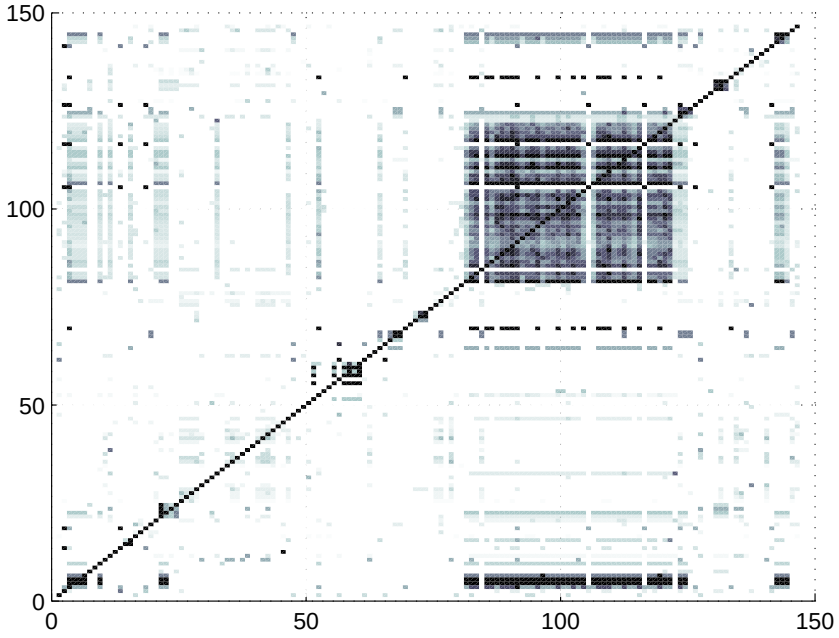


Figure 4.5: Developers' overlap in the Apache ecosystem in 2013.

velopers already active on other (previously existing) projects. A relevant example is represented by the APACHE COMMONS projects, that are evolved and maintained by a very cohesive community of developers. The number of distinct developers working in 2013 on the Apache ecosystem is 147, against 2674 developers counted by project (i.e., a developer is counted multiple times if she works on more than one project). However, if we count, for each APACHE COMMONS projects, the number of developers working on it (i.e., a developer is counted multiple times if she works on more than one project) we obtain a total of 2,674 developers. This means that there is a strong overlap of developers between these projects. Thus, when new projects are added to the ecosystem, this does not necessarily imply that new developers also join the ecosystem.

To get a better view of the developers' overlap existing between the Apache projects, Fig. 4.5 shows, for each pair of the analyzed 147 Java projects, the percentage of developers overlap in 2013: black means 100% of overlap between the two projects, white means 0% of overlap. The diagonal is colored in black by default, since each project will have 100%

of developers' overlap with itself. As we can see, several projects share developers, also in high percentage. The black rectangle that can be observed in the right-up part of Fig. 4.5 corresponds to the APACHE COMMONS projects.

While the overlap figure explains the stable number of developers between 2006 and 2011, it is still unclear why from 2001 to 2013 the developers base decreases, despite the increase of projects in the ecosystem. As it can be noticed from Fig. 4.3, on the one hand the number of projects does not have a substantial increase between 2011 and 2013 (12 projects added). On the other hand, in such a period the overall ecosystem LOC (Fig. 4.2) increased of about 18%. That is, changes occurred in the ecosystem mainly concerned addition/improvement of features in existing projects (as well as bug fixes). However, this kind of activity concerned a relatively limited number of developers, whereas many developers worked on the early development activity of each project. Also, until 2011, several projects have reached a very stable state, in which there is not a lot of activity. Therefore, we see a decrease in the number of active developers (which does not necessarily mean that developers abandon the project). For example, the last release of APACHE CHAINSAW[12] is dated March 2006, while the last release of APACHE COMMONS BETWIXT[13] is dated March 2008.

Referring to the previous examples, the number of developers in APACHE CHAINSAW has decreased from eight in 2004 to one in 2013, while the number of developers in APACHE COMMONS BETWIXT from 19 to five in the same time period. Besides that, we noticed that also very active projects—i.e., still issuing releases during the last year—had a decrease of active developers due the reached mature state. For instance, in 2005 APACHE COCOON[14] had 64 active developers; nowadays the number of active developers is reduced to seven. Finally, it is worth noting that Goeminne *et al.* [94] observed in the GNOME ecosystem a variation trend for the number of active developers very similar to the one we found in the Apache ecosystem. Specifically, after an initial increase of active developers from 1997 to 2003, they found the developers base to be almost stable until the 2008. Then, they observed a decrease of the active developers from 2008 until 2013.

Fig. 4.3 shows that the number of dependencies between projects continuously increases during evolution. Similarly to the size, but differently from the number of projects, dependencies follow an exponential trend (adjusted $R^2 = 0.56$). In fact, until 2003 (when about 25 projects were in the ecosystem) there were few dependencies between the projects. After 2003, dependencies sensibly grow in the following years. This is mainly due to the fact that several projects added after 2003 are projects implementing reusable components—like

[12] `http://logging.apache.org/chainsaw/`
[13] `http://commons.apache.org/proper/commons-betwixt/`
[14] `http://cocoon.apache.org/`

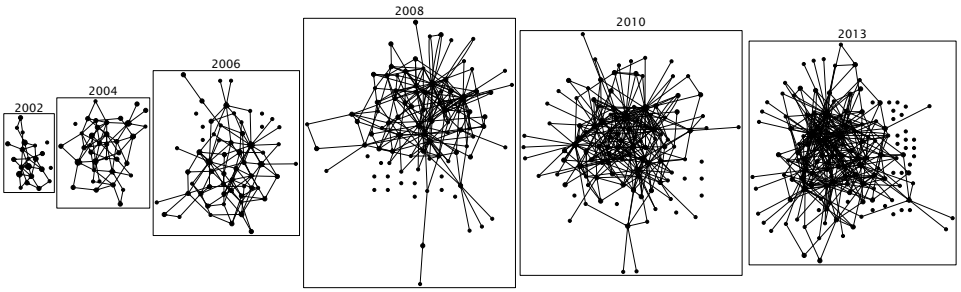


Figure 4.6: Snapshots of projects and their dependencies in the Apache ecosystem history.

those belonging to the APACHE COMMONS[15]—that are used as libraries by several Apache projects. For example, the number of client projects for APACHE COMMONS COMPRESS[16] grows up to 20 (April 2013).

To provide a better view on how the Apache software projects and the dependencies between them evolved during time, Fig. 4.6 shows snapshots of the Apache ecosystem from 2002 to 2013. We ignored the years before 2002 since, as reported in Fig. 4.3, the number of projects (and dependencies) is quite low. In the graphs of Fig. 4.6, each node represents a project, while an edge connecting two nodes represents a dependency between two projects. By looking at the figure it is clear as the net of dependencies in the ecosystem grows during evolution. Also, if focusing on the 2013 snapshot, we can notice several *hub projects*, i.e., projects having a lot of client projects. Besides the previously discussed APACHE COMMONS, other *hub projects* are for example APACHE LOG4J[17] (having 31 client projects), APACHE GERONIMO[18] (30), and APACHE ANT[19] (29). It is worth noting that all these projects implement quite general and reusable features, useful for software projects having different purposes.

RQ_1 summary: We can summarize results of $\mathbf{RQ_1}$ stating that the Apache ecosystem size and dependencies exponentially increase over time. Instead, the number of active developers increased until a certain point (2006), then it remained stable until 2011, since new projects were basically maintained by existing developers, and finally decreased because some projects became stable and required less activity.

[15] http://commons.apache.org/
[16] http://commons.apache.org/proper/commons-compress/
[17] http://logging.apache.org/log4j/
[18] http://geronimo.apache.org/
[19] http://ant.apache.org/

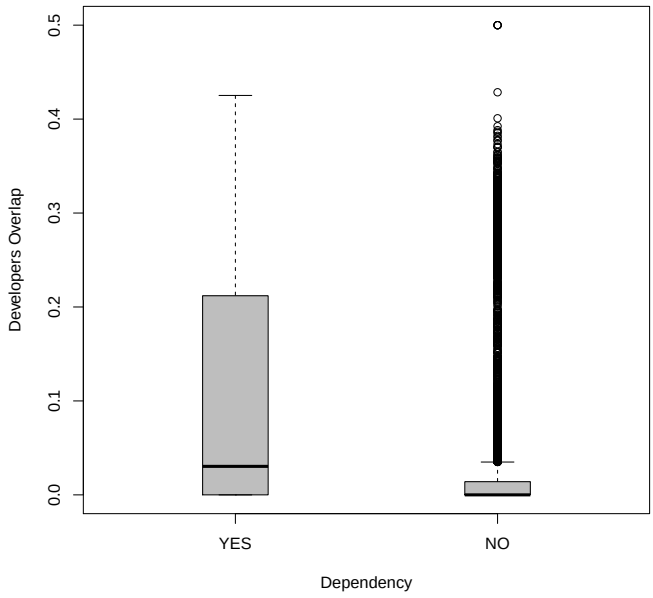


Figure 4.7: Developers' overlap (in percentage) in projects having and not having a dependency.

4.3.2 RQ_2: What is the relation between sub-projects developers overlap and sub-projects dependencies?

In the following, we report and discuss results for each one of the independent variables of $\mathbf{RQ_2}$ described in Section 4.2.2.

Developers' overlap. Concerning the overlap of developers between projects, we first investigated whether pairs of projects having a dependency share a greater number of developers than projects do not having a dependency. Fig. 4.7 reports the distribution of developers' overlap between projects having and not having a dependency, showing as the former generally share a higher number of developers as compared to the latter. This difference is also statistically significant: the Mann-Whitney test returned a p-value <0.01, with a medium effect size (Cliff's d=0.47).

After that, we checked whether among the pairs of projects having a dependency, those sharing a higher number of developers with the library they use also exhibit a higher upgrade

Table 4.2: Tags manually assigned to the 871 discussions talking about dependencies between projects.

Tag	Number of discussions	Percentage
GENERAL TAGS		
client	759	87%
library	112	13%
DEVELOPERS TAGS		
only developers client	725	83%
only developers library	107	12%
both developers	39	5%
TOPIC TAGS		
break	24	3%
fix	283	33%
upgrade	187	22%
use	53	6%
other	324	36%

frequency by the client (i.e., the client tends to upgrade more frequently to new releases of the library projects). In particular, we computed the Spearman correlation between the client upgrade frequency and the average overlap of developers it has with its libraries. However, we only observed a small correlation (ρ =0.13, p-value<0.01).

RQ_2 summary: Pairs of projects having a dependency share a greater number of developers than pairs of projects not having a dependency. This result is a first indication that client and library projects co-operate in the management of the dependency, that will be object of our **RQ_3**.

4.3.3 RQ_3: How are dependencies discussed between developers?

Among the 7,685 discussions manually analyzed, 871 received the *dependency* tag, indicating that the discussion was actually about the management of a dependency between a client and a library project. Table 4.2 reports, for each of the tags considered in our study, the number (and percentage) of discussions to which it has been assigned.

Starting from the "general tags", it is clear that most of the discussions on dependencies' management is carried out on the client projects' side. In fact, 759 out of the 871 discussions

(87%) were extracted from the clients mailing lists and issue trackers. This is an expected result, since it is reasonable to think that between client and library projects the former ones are those more interested in the correct working of dependencies. This is also confirmed by the "developers tags" showing as 83% of discussions related to dependencies only involve developers from the client project. However, even if in a smaller proportion (12%), also developers of the library projects discuss about dependencies management, sometimes together with the developers of the client projects (5%).

Concerning the topic object of the discussions ("topic tags" in Table 4.2), we found 33% of them focused on fixing problems caused by a dependency (tagged as *fix*). Specifically, we observed discussions concerning different kinds of problems. The most common problems observed are those related to bugs present in the used library, consequently causing a bug in the client project. For instance, such a kind of problem was discussed in the APACHE STANBOL[20] mailing list. APACHE STANBOL is a software providing reusable components for semantic content management (e.g., automatic tag extraction from webpages, text completion in search fields, etc.), and it uses as library APACHE TIKA[21], a toolkit able to detect and extract metadata and structured text from various document formats. Developers of the client project discuss[22] about problems related to the extraction of metadata from JPEG images. This feature is provided to STANBOL by the TIKA library.

Another discussion tagged as *fix* occurred in the APACHE MINA[23] mailing list. MINA, a network application framework to develop high performance and scalability networks, is used as library by the APACHE SSHD[24] project, supporting SSH protocols for client-server communications. In this case[25], developers are discussing about a problem found MINA 2.0.2, and causing a bug in SSHD. The solution has been the simple upgrade to MINA 2.0.4, fixing the reported issue.

22% of the analyzed discussions was tagged as *upgrade*, indicating that the discussion was focused on whether upgrading or not a dependency towards a new available release of a library project. A very interesting example is the one we found in the APACHE TORQUE[26] mailing list. TORQUE is an object-relational mapper for java using several Apache projects as library (i.e., COMMONS BEANUTILS, COMMONS COLLECTIONS, COMMONS CONFIGURATION, COMMONS LANG, XERCES-J, XML COMMONS, VELOCITY, and ANT). Developer

[20] http://stanbol.apache.org/
[21] http://tika.apache.org/
[22] http://tinyurl.com/p3nxkyc
[23] http://mina.apache.org/
[24] http://mina.apache.org/sshd-project/
[25] http://tinyurl.com/nfrqfrf
[26] http://db.apache.org/torque/torque-4.0/index.html

T.F. wrote in the discussion[27]:

> *I have gone through the libraries Torque depends on and seen if there is a newer version available. Those are the updates I would suggest:*
>
> *commons-beanutils from 1.6.1 to 1.7.0*
> *commons-collections from 3.0 to 3.1*
> *commons-configuration from 1.0 to 1.1*
> *commons-lang from 2.0 to 2.1*
> *xercesImpl from 2.4.0 to 2.6.2*
> *xml-apis from 1.0.b2 to 2.0.2*
> *ant from 1.5.1 to 1.6.5*
>
> *[...]*
>
> *Note that velocity is not updated to 1.4. I have heard rumors that version 1.4 has a memory leak (but I have also heard rumors that the current velocity version chokes in very large files, so not sure whether the memory leak is not already there in 1.3.1). [...]*

This discussion suggests that (i) sometimes the choice to ignore a new available release of a library the client depends on (VELOCITY 1.4 in this case) is based on the fear to introduce errors in the client project, and (ii) even when some effort is spent to upgrade dependencies like in this case, not all dependencies are upgraded together.

Several discussions tagged as *upgrade* also confirmed the fact that potential brakes in the client push away the client from upgrading their dependencies. For instance, in the APACHE ROLLER[28] mailing list we found a discussion [29] on whether upgrading or not a dependency towards APACHE LOG4J. In particular, one of the ROLLER developers asked if it is the case to upgrade the release of LOG4J used in ROLLER:

> *log4j is up to 1.2.12. We're still using/distributing 1.2.4 in the trunk (bound for 2.0). I think we should upgrade to 1.2.12 in the trunk.*

The answer came from another ROLLER developer:

[27] http://tinyurl.com/qjyw6u2
[28] http://roller.apache.org/
[29] http://tinyurl.com/qz254l8

> *I recently tried to upgrade to 1.2.12 and found that there were some incompatibilities with my config file. I forget what they were - but it basically wasn't a simple upgrade. For that reason, I'm currently using 1.2.11.*

Thus, even if the ROLLER developers were going to issue their new release 2.0 and were conscious of using an old LOG4J release, the choice was to not risk to perform a tricky upgrade.

The *use* tag has been assigned to 6% of the analyzed discussions, dealing on how to use the library in the client, while only 3% of them were tagged with the *break* tag, indicating discussions aimed at avoiding changes that could break the dependency. These discussions generally happen in the library projects' communication channels. For instance, in the APACHE GERONIMO mailing list we found a discussion[30] where a developer was alerting about the possible issues that could be caused by the removal of a dependency in the project: *This change removed Sun SAAJ implementation dependency. That dependency is currently needed and should not be removed (I'm pretty sure it will break CXF)*. Note that APACHE CXF[31] is a client project using GERONIMO and, among the developers of these two projects, we found a dense network of communication mostly carried out by developers in overlap between the two projects. We depicted this network in Fig. 4.8, where CFX's developers are represented with blue nodes, GERONIMO's developers with orange nodes, and developers in overlap between the two projects are colored in yellow. An edge exists between two developers if they exchanged at least two messages (i.e., the communication between them is not occasional). Blue edges are messages exchanged in the CFX's communication channels, while the red ones are messages exchanged in the GERONIMO's communication channels. From Fig. 4.8 it is interesting to notice that (i) most of the developers in overlap are hubs exchanging messages with several other developers, (ii) most of the communications between the two projects passes through the developers in overlap (i.e., the yellow nodes).

Finally, 36% of discussions were tagged as *other*, because they were related to topics that cannot be placed in the previous discussed tags. Examples are discussions related to missing references in the release bundles, to the possibility of whether or not providing support to old releases of some clients, or to legal issues. An example was a discussion started by an APACHE APERTURE developer:

> *Hello Tika!*
> *Hello Aperture!*

[30] `http://tinyurl.com/opmlu8z`
[31] `http://cxf.apache.org/`

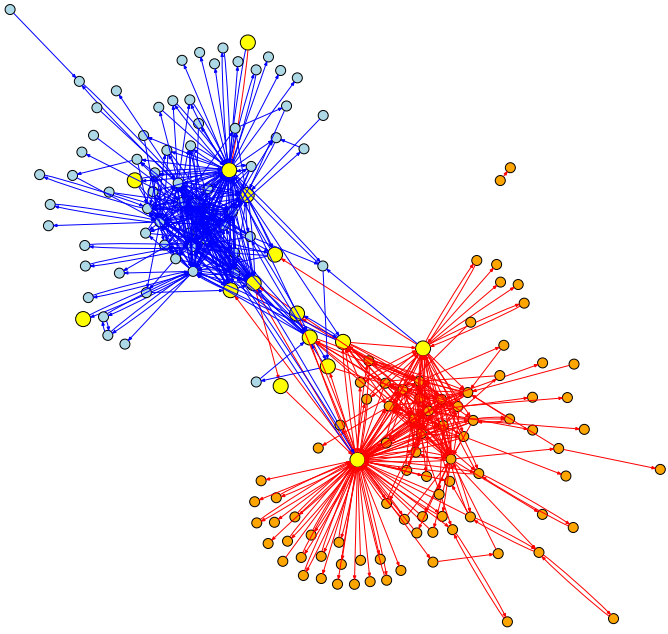


Figure 4.8: Communication network between Geronimo and CFX developers. CFX's developers are shown in blue, Geronimo's developers in orange, while yellow circles are developers overlapping between the two projects.

> *We (the Aperture project) have recently updated the pdfbox to the current trunk version. It seems that they've introduced a new dependency on the Java Advanced Imaging API (JAI). The problem is that JAI imposes certain constraints on redistribution. [...]*

RQ_3 summary: The manual analysis performed in the context of RQ_3 showed that developers actively discuss about dependency management. Generally, this discussion is carried out by developers of the client projects mainly discussing about problems due to the (*fix*) of dependencies, and of the possibility to whether or not upgrading a dependency.

4.4 Threats to Validity

This section discusses the threats that can affect the validity of the achieved results.

Threats to *construct validity* concern the relation between the theory and the observation. They can be mainly due to imprecisions in the measurements we performed. This is a

summary of the main sources of imprecision:

- The mapping between dependencies declared within a project and other projects was performed using a set of heuristics, as explained in Section 4.2.2. To cope with the imprecision of such heuristics, results were manually verified.
- In the analysis of the evolution of active developers in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we cannot exclude that the projects also involved other contributors whose activity is not evident from the versioning system commits.
- The analysis of the nature of changes performed in $\mathbf{RQ}_2$ involved a manual classification of releases. This could have lead to some subjectivity in the classification. To avoid that, two of the authors performed the classification independently, and then discussed cases where their choices were inconsistent.
- The analysis of developers' communication performed to address $\mathbf{RQ}_3$ has been conducted by considering, as communication means, project mailing lists and issue reports. In many projects—and especially in worldwide-distributed open source projects like the ones we analyzed—it is a consolidated practice to communicate through mailing lists and issue trackers. However, we are aware that there could still be some hidden communication [34] we might have missed in our analyses. A different matter concerns the manual tagging of such a communication which, due to the large number of emails/issues to be analyzed (7,695), was split between the two inspectors. Although we are aware that mistakes could have occurred, both inspectors agreed on guidelines to perform a classification, and they discussed together unclear cases.

Threats to *internal validity* concern factors internal to the study that could influence our results. Such kind of threats typically do not affect exploratory studies. The only case worthwhile of being discussed is about $\mathbf{RQ}_2$. In the first case, although we have found some correlation between certain kinds of changes and upgrades decisions, we cannot claim there is a cause-effect relation. In addition to that, the large manual analysis of developers' communication conducted in the context of $\mathbf{RQ}_3$ provided a strong qualitative support to the quantitative findings of $\mathbf{RQ}_2$.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. The analyses performed in this study mainly have an observational nature, although we used, where appropriate ($\mathbf{RQ}_2$), statistical procedures and effect size measures to support our claims.

Threats to *external validity* concern the generalizability of our findings. Such a generalizability is clearly limited to the ecosystem being analyzed, i.e., Apache, and specifically Java

projects of the Apache ecosystem. Also, in terms of assessing dependency upgrades, such assessment is confined to *within-ecosystem* dependencies, as we are not interested to analyze dependencies to projects that are not part of the ecosystem. Future studies need to be done to investigate upgrades with respect to external dependencies too, and to repeat the study on other ecosystems.

4.5 Related work

In this section, we discuss the related literature, focusing our attention on (i) work studying software ecosystems and (ii) work observing the impact on software evolution and stability of changes/deprecations of APIs.

4.5.1 Analysis of Software Ecosystems

In the last decade several software ecosystems have been studied from different perspectives. Table 4.3 reports these studies, classifying them by (i) the ecosystem being studied, (ii) the source of information exploited, (iii) the objectives of the study, and (iv) the main findings.

One of the first software ecosystems subject of several empirical studies has been the Debian Linux distribution [96–98]. Specifically, Godfrey *et al.* [96] analyzed the size of the Linux operating system Kernel, observing a super-linear rate growth for several years. Gonzalez-Barahona *et al.* [97] found that the Debian Linux distribution has been doubling in size every two years while the average size of its packages remained stable over time. Also, they observed as the number of dependencies between packages increased exponentially. German *et al.* [98] proposed a methodology and visualization tool aimed at supporting the study of inter-dependencies in complex software systems. The tool has been used to analyze the dependencies between projects in the Debian Linux distribution. Capturing dependencies between projects in an ecosystem is far from trivial [111] and it is the reason why several authors focused their attention on methods for the extraction of dependencies in large software ecosystems [98, 99]. Similarly to what done in other studies, in the context of our work we also exploited specific heuristics (see Section 4.2.2) to identify the dependencies between projects.

Another software ecosystem that has been studied by several authors is the Eclipse IDE. Wermelinger *et al.* [100, 101] analyzed the evolution of the Eclipse's architecture and found that the success as application framework for the Eclipse SDK mainly depends from the fact that it *"follows several practices that support sustainable architectural evolution"* [101]. In particular, the Eclipse developers manage APIs carefully, avoiding to break existing APIs

Source	Objectives	Findings	Ref
	APACHE ECOSYSTEM		
Versioning System and Mailing lists	Defining a set of invariant metrics to detect "stagnant projects".	Stagnant projects can be identified by measuring the ratio of the e-mail exchanged in mailing lists and the number of commits.	[95]
Versioning System, Release Notes, Issue Trackers, and Mailing lists	Focus on projects dependency management.	Clients tend to upgrade their dependencies when libraries are subject to bug fixes, while changes to interfaces make the upgrade less appealing. Most of the times the impact of upgrades is well-confined.	**Our Work**
	LINUX ECOSYSTEM		
Versioning System	Analysis of the evolution of the Linux Kernel.	The size of the Linux Kernel has been growing at a super-linear rate for several years.	[96]
Versioning System	Analysis of the evolution of the Debian Linux distribution	The overall size has been doubling every 2 years, while the average size of packages remained stable. Instead, the number of dependencies increased exponentially.	[97]
Versioning System	Analysis of dependencies in the Debian Linux distribution.	A methodology and visualization for studying inter-dependencies of a complex software system.	[98]
	SQUEAK ECOSYSTEM		
Versioning System	Recover dependencies between the software projects of the Squeak ecosystem.	Accurate detection of dependencies for Smalltalk.	[99]
Versioning System	Analysis of API changes in a software ecosystem.	API changes caused by deprecation can have a very large impact on the ecosystem in terms of the number of changes needed to fix broken dependencies.	[88]
	ECLIPSE ECOSYSTEM		
Versioning System	Analysis of the Eclipse architecture.	The development follows a systematic process and there is a stable architectural core that remains since the first release.	[100, 101]
Versioning System	Analysis of the evolution of Eclipse core plugins.	Eclipse plugins adhere to the laws of continuing change and growth, but not to the law of increasing complexity.	[102]
Versioning System	Analysis of the evolution of third-party plugins.	Third-party plugins that depend from stable and supported Eclipse APIs have a higher compatibility success rate than plugins depending on discouraged and unsupported APIs.	[90]
Mailing lists and issue trackers	Analysis of developers' productivity.	Adding new features to Eclipse slows down the bug fixing process.	[103]
	GNOME ECOSYSTEM		
Versioning System	Analysis of the active developers.	The number of active developers increased until 2003, remained stable until the 2008, and then decreased.	[94]
Versioning System	Analysis of the ecosystem evolution.	A list of practices that could benefit both open and commercial software development organizations.	[104]
Mailing lists and issue trackers	Analysis of the activities on and contributors of the software ecosystem.	GNOME contains both paid contributors and volunteers. Coding is the most preeminent activity in the ecosystem.	[105–107]
Versioning System	Analysis of cloning and copying operations between GNOME Projects.	Larger clones exist between the sub-projects of GNOME and more than 60% of the clone pairs can be automatically separated into original and copy.	[98]
Versioning System, Mailing lists and Issue Trackers	Analysis of social processes in the ecosystem.	Participants in such ecosystems may be able to use a significant amount of transferrable knowledge when moving between projects in the ecosystem and, thereby, skip steps in the "onion model"	[108]
	GNU R ECOSYSTEM		
Versioning System and Mailing lists	Analysis of the differences between code characteristics of core and user-contributed packages.	User-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages.	[91]
	GURUX ECOSYSTEM		
Versioning System and Mailing lists	Analysis of the evolution.	Supporting processes, guidelines and best practices for building open source communities.	[109]
	FIREFOX, UNITY, AND	**GOOGLE CHROME ECOSYSTEMS**	
Versioning System	Analysis of how the software licenses are reported in software ecosystems.	Software component licenses and the architectural composition of a system help to better define the software ecosystem "niche" in which a given system lies (i.e., the license is wrong).	[110]

Table 4.3: Studies that analyzed software ecosystems.

when issuing new releases. Mens *et al.* [102] found that the Eclipse core plugins adhere

to the laws of continuing change and growth, but not to the law of increasing complexity. Businge *et al.* [90] analyzed the dependencies and the survival of 467 Eclipse third-party plugins. They found that plugins depending only on stable and supported Eclipse APIs have a very high source compatibility success rate. This means that third-party plugins that depend from stable and supported Eclipse APIs have a higher source compatibility success rate than plugins depending on discouraged and unsupported Eclipse non-APIs. In addition, Businge *et al.* found that the majority of plugins hosted on SourceForge[32] do not evolve beyond the first year of release.

GNOME is another very well investigated software ecosystem [94, 104–107, 112]. German *et al.* [104] distilled a list of practices that could benefit both open and commercial software development organizations by studying the GNOME ecosystem. For example, a careful coordination of the development activities between the sub-projects belonging to the ecosystem can be the one of the keys for the success of the different projects. Mens *et al.* [105, 107] studied the GNOME mailing lists and issue trackers observing that GNOME contains both paid contributors and volunteers. Also, coding seems to be the most preeminent activity in the ecosystem, followed by activities such as translation and development documentation. In addition, members of the GNOME community tend to specialize themselves in a limited number of activity types [105]. Goeminne *et al.* [94] observed in the GNOME ecosystem a variation trend for the number of active developers similar to the one we identified in the Apache Ecosystem. Specifically, after an initial increase of active developers in the 1997-2003 time window, they found the developers base to be almost stable until 2008. Then, they observed a decrease in the number of active developers up to date.

German *et al* [91] studied the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that the ecosystem of user-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages, yet each individual package remains stable in size.

Scacchi *et al* [110] examined how the software licenses are reported in software ecosystems and in particular, observe how software component licenses and the architectural composition of a system help to better define the software ecosystem "niche" in which a given system lies (i.e., the license is wrong).

Other than the analysis of ecosystems evolution, social/community aspects of ecosystems (for example bug reports and/or mailing list traffic between developers teams) have also been analyzed [103, 106, 108, 109]. Kidane *et al.* [103] found that adding new features to Eclipse slows down the bug fixing process.

[32] `http://sourceforge.net/`

Yu *et al.* [113] studied the mailing lists of the Linux kernel, to analyze different ecosystem collaboration patterns between companies. Jergensen *et al.* [108] instead, analyzed multiple systems which have "common underlying components, technology, and social norms". They observed how participants in such ecosystems may be able to use a significant amount of transferrable knowledge when moving between projects in the ecosystem and, thereby, skip steps in the "onion model"[33]

Annosi *et al.* [114] proposed a framework to support developers in the upgrade of third-party components. The decision is driven by various factors, partially related to the kind of change occurred in the component (as mined from release notes or issue trackers), and partially on expert judgements collected within the company.

Gala *et al.* [95] also analyze the Apache ecosystem proposing a set of "invariant metrics" in the domain of software projects. They found that metrics measuring the proportion (or ratio) of the e-mails exchanged in mailing lists and the total number of commits performed by developers can be useful to identify stagnant projects and projects in danger of stagnation. In our study, we also observed that project's stagnation is one of the factors that reduces the active developers' base.

4.6 Summary

This study investigated on the evolution of project inter-dependencies in the Java subset of the Apache ecosystem, comprising a total of 1,964 releases of 147 projects, for 14 years. First, we investigated how the ecosystem has grown over time in size, number of projects, dependencies between projects, and number of developers. After that, we analyzed the factors that could have influenced the upgrade (or not) of a dependency between a project and a library it uses. Also, we qualitatively investigated, by looking into mailing list and issue report discussions, how developers discussed the opportunity to perform an upgrade and its possible impacts/risks.

The study results indicated that:

- The ecosystem size exponentially grows over time, and consequentially the dependencies between projects grow too. The number of active developers involved in the project grows until a certain year (2006). Then, it remains stable because most of the new projects (many of which part of the APACHE COMMONS) involve developers already active in the ecosystem). Finally, over the last few years (since 2011) we observe

[33] The onion model is a socialization process where newcomers join a project by first contributing through mailing list discussions and bug trackers and they advance to more important roles contributing where they can improve the code and making design decisions.

a decrease in the number of active developers. This can be explained because very few projects have been added to the ecosystem during such a period, while the size growth is mainly due to the evolution of existing projects. The latter could, however, concern only some specific features and therefore be performed by a subset of developers only.

- For what concerns the factors that could have influenced dependency upgrades, we found that this does not really depend on project-level characteristics such as project size. Moreover, pairs of projects having a dependency share a higher number of developers as compared to pairs of projects do not having a dependency. These findings have been reflected from the discussions developers had over mailing lists and issue trackers.

This work has mainly an observational nature, i.e., it aimed at empirically investigating a phenomenon—dependency upgrades in a software ecosystem—-from both quantitative and qualitative point-of-view. Nevertheless, there are different possible uses one can make of the results of this study. First, the study highlights that the dependency phenomenon has an exponential growth and should therefore carefully be considered by developers contributing to the ecosystem. Second, it provides an overview of possible factors that could influence dependency upgrades, with indications of the role played by such factors in the Apache ecosystem, and of how the main reasons for upgrading or not were discussed by developers. This means that developers discussions are massively characterized by dependencies discussion of dependent sub-projects. Such information can be used to define an approach to avoid changes that could break the dependency with third-party libraries.

Part II

How Developers Browse and Understand Software Artifacts

Code comprehension activities often start by reading source code, and continue by abstracting away and browsing software artifacts at higher level of abstraction [115]. This part of the thesis investigates, at different levels of abstraction, how developers perform comprehension tasks. Specifically, in Chapter 5 we analyze navigation patterns developers follow across use cases, class diagrams, sequence diagrams, and source code. This analysis can be useful to build recommenders to better navigate the documentation during maintenance activities. Results indicate that, although newcomers spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and class/sequence diagrams. Less frequently, developers—especially more experienced ones—follow an "integrated" approach by using different kinds of artifacts. Such information can be seen as a starting point to built recommenders to better navigate the documentation during maintenance activities. When it comes to understanding source code, in Chapter 6 we investigate how developers summarize different code elements, i.e., what terms do they use. Specifically, we investigate whether the words used to summarize a class are high-frequency words, or belong to a high-frequency concept. Also, we investigate what kinds of terms do developers use, e.g., comments, words from method signatures, attribute names, etc.

Chapter 5

An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks

Contents

In the Chapter 4 we investigated (i) how sub-projects evolve over time when an ecosystem grows, (ii) what are the product and process factors that can likely trigger dependency upgrades and (iii) how developers discuss the needs and risks of such upgrades.

Hence, software ecosystems consist of multiple software projects, often interrelated by means of dependency relations. When one project undergoes changes, other projects may decide to upgrade their dependency. For example, a project could use a new version of a component from another project because the latter has been enhanced or subject to some bug-fixing activities. During this process developers teams between sub-projects of the ecosystem discuss and manage dependencies in order to avoid bugs, and release new version in easier way.

However, when newcomers join a software project and perform a software maintenance task, they need to identify artifacts—e.g., classes or more specifically methods—that need to be modified. To this aim, they can browse various kind of artifacts, for example use case descriptions, UML diagrams, or source code.

This Chapter reports the results of a study—conducted with 33 participants—aimed at investigating (i) to what extent newcomers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts.

Results indicate that, although newcomers spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, developers—especially more experienced ones—follow an "integrated" approach by using different kinds of artifacts. Such information can be seen as a starting point to built recommenders in help newcomer to choice appropriate patterns in navigate software documentation when apply maintenance tasks.

5.1 Motivation: help newcomers to properly navigate documentation during maintenance activity

Maintenance tasks are generally facilitated when software documentation (e.g., the requirements specification, design document, test report, and user manual) is available [116–118]. Indeed, having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20% [118]. In addition, besides time reduction, documentation allows developers to find better and more accurate technical solutions to a given maintenance task [118].

Although several studies have shown the usefulness of documentation during maintenance tasks (see e.g., [116–121]), it is still unclear how such documentation is browsed by developers to understand how the system should be modified to implement a specific change. At one extreme, one can argue for using all the available documentation, as each artifact is equally useful, since it provides a description of the system with different levels of details. Also, the documentation could be browsed starting from HLA (e.g., use cases) to LLA (e.g., dynamic models). Even if there is an anecdotal evidence that such an approach could work, without a proper empirical investigation it remains only a conjecture. Also, different developers—with different skills and experience—might follow different paths. Thus, on one hand, guessing a priori navigational paths is quite challenging. On the other hand, understanding such paths is relevant not only to highlight the importance of high-level documentation, but also to help tool developers enhancing modelers and Integrated Development Environments (IDEs) to better support program comprehension activities by facilitating effective and efficient artifact navigation and browsing.

All these considerations motivate our work. We conduced a study, involving 33 participants (among undergraduate and graduate students from different universities) aimed at analyzing to what extent developers use different kinds of documentation when identifying pieces of code (e.g., methods) to be changed and whether they follow specific navigation paths among different kinds of artifacts. In the context of our study, we asked participants to perform 8 different maintenance tasks on a Java software system. It is important to note that the subjects considered in our study are not familiar with source code because we want to simulate the scenario in which newcomers that are joining a software project are applying a maintenance tasks. Besides source code, participants had available use case descriptions, sequence diagrams, class diagrams, and Javadoc. We used an Eclipse plugin to capture how much time was spent by participants on different artifacts, and how they navigated from an artifact to another.

The obtained results indicated that—even if a substantial proportion of time (about 80%

on average) is spent on source code, participants also browsed back and forth between source code and either static (class) or dynamic (sequence) diagrams, the latter being more used than the former. Less frequently, participants—and in particular those with a higher degree of experience, i.e., graduate students—follow an "integrated" approach, in which different kinds of artifacts were used, for example starting the task from use cases, then browsing sequence and/or class diagrams before accessing the source code. Such results could be used to enhance IDEs with a recommendation system able to suggest a particular navigation path aiming at facilitating the browsing of the available documentation. Such a recommender might be particular useful in large systems where the browsing of myriad software artifacts could represent an obstacle instead of a facilitation when performing the maintenance task [122, 123].

This Chapter is organized as follow. Section 5.2 presents the definition and planning of our study, while Section 5.3 discusses the results achieved. Section 5.4 presents the threats that could affect the validity of our study. Finally, after a discussion of the related literature (Section 5.5), Section 5.6 summaries the results of the empirical study.

5.2 Study Definition and Planning

This section describes the design and planning of the our empirical study. The *goal* of the study is to observe how developers browse different kinds of software artifacts, with the *purpose* of understanding how they build knowledge needed to deal with a maintenance task and, specifically, to identify classes and class elements (methods and attributes) that need to be changed when performing a maintenance task. The *perspective* is of researchers interested to identify relevant navigation paths across artifacts that result helpful during a software evolution task. This result can be used, for example, to build smart recommenders that guide developers by suggesting navigations across artifacts or to better organizing and indexing the documentation available for a software project.

5.2.1 Context Selection

The study involved 33 participants, selected entirely on a voluntary basis—i.e., using a convenience sampling—mainly among undergraduate students of the Computer Science Degree at the University of Molise, and among master students, PhD students (including visiting students) of the Computer Science Engineering Degree of the University of Sannio. Overall, 11 Bachelor students, 18 Master students, and 4 PhD students participated to the study.

The objects which the tasks were performed on are use case descriptions, design level sequence and class diagrams, Javadoc, and Java source code files of a school automation

system, named SMOS, developed by graduate students at the University of Salerno (Italy). SMOS offers a set of features aimed at simplifying the communication between the school and the student's parents. The system is composed of 121 classes with their respective Javadoc for a total size of 23 KLOC. The documentation is represented by 67 use cases, 72 design level sequence diagrams, and 6 design level class diagrams. Each class diagram represents the relationships between all the classes involved in a specific subsystem, e.g., teaching management.

In the context of our study, we asked participants to perform 8 different maintenance tasks on SMOS, of which 3 were bug-fixing tasks, 3 related to add a new feature, and 2 related to improve existing features, i.e., performing a perfective maintenance task.

5.2.2 Research Questions

The study aims at investigating the following research questions:

- **RQ$_1$:** *How much time did participants spend on different kinds of artifacts?* This research question aims at analyzing how much time participants spent on different kinds of artifacts. On the one hand, artifacts used for less time can be thought of being less useful. On the other hand, some artifacts intrinsically require more time to be read (e.g., source code) while for others (e.g., use cases, sequence diagrams) a quick look may just suffice to provide a useful piece of information.

- **RQ$_2$:** *How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?* This research question is the core of our study aimed at analyzing the sequences of interactions made with different artifacts. In particular, we will investigate (i) how do participants start the task, (ii) what kinds of artifacts do they browse before getting to the source code, and (iii) whether there are frequent browsing patterns, e.g., repeated navigation back and forth between source code and class diagrams.

For each research question, we also analyzed the impact of participants' experience on the use and the navigation of the software documentation.

5.2.3 Study Procedure and Material

Before the study, we explained to participants what we expected them to do during their tasks. Specifically, we asked them to identify methods and attributes to be changed when performing each change task. We provided an overview of what kinds of artifacts they have available, briefly summarizing the purpose of each of them.

TASK DESCRIPTION

In SMOS a registered user can have six different roles: Admin, Teacher, Student, Parent, Janitors, and Director. Suppose that we want to remove the "Director" role, which changes do you need to made on source code? Specify for each involved class/method the changes you would apply.

QUESTIONS

Write the list of methods modified to perform this task specifying how you modified these methods. For example: "application.userManagement.UpdateUser.doGet". I added the line of code "x=3;" after the line of code "y++;".

Write the list of attributes modified to perform this task.
For example: "bean.User.UID".

Figure 5.1: Example of task description and related questions.

After illustrating the study, we gave participants up to 3 hours of time to perform the task. Note that it was not our intention to measure the task efficiency, hence we were not strict with the time. We only made sure participants properly performed the task, without collaborating.

We provided each participants with a customized Eclipse installation containing:

- The Java Development Environment (JDT) with the SMOS software system already imported together with its documentation, i.e., sequence diagrams, class diagrams, use cases, and Javadoc.

- FLUORITE (Full of Low−level User Operations Recorded In The Editor[1]), an Eclipse plug-in able to capture all of the low-level events when using the Eclipse editors. FLUORITE keeps track of all of the events that occur in the Eclipse editors also storing timestamps for each event. All data is saved in an XML log file.

- The Pdf4Eclipse[2] plug-in (used to visualize use cases, sequence diagrams, and class diagrams).

[1] http://www.cs.cmu.edu/ fluorite/
[2] http://borisvl.github.io/Pdf4Eclipse/

- An Eclipse HTML Editor[3] plug-in, used to visualize the Javadoc files.

Also, we provided participants with an URL of a page on ESurveysPro[4], a online survey tool we used to collect participants' answers.

During the study, we instructed participants to access the ESurveysPro page and, for each of the eight tasks to be performed, to work following this procedure:

[1] access the page describing the task, and read the task description

[2] then, use Eclipse to find a solution for the task

[3] after the task has been performed, answer the questions in the opened ESurveysPro page. For each task, participants had to provide, using two different form fields, the list of methods and instance variables (attributes) that need to be modified. Fig. 5.1 shows an example of task description and questions being asked for the task. For each question examples of answers are provided. We made clear to participants that example answers are not related to the task, thus they are not valid answers.

After having completed the 8 tasks, participants had to fill a post-study questionnaire. The post-study questionnaire asked participants an opinion about the usefulness of the various kinds of artifacts, using a Likert scale [124] ranging between 1 (totally useless) and 5 (very useful). We also asked participants to provide a comment for the rank assigned to each kind of artifact.

5.2.4 Data Collection

After tasks were completed, we collected from each participant (i) the XML logs generated by FLOURITE; and (ii) the answers provided on ESurveysPro. Concerning FLUORITE logs, they have been parsed through a Java tool developed on purpose. The tool extracts, for each task performed by each participant, the ranked list of documents explored during such a task together with the time spent on each document. An example of generated list is:

$$UseCase(27) \rightarrow SequenceDiagram(48) \rightarrow Code(82)$$

indicating that the participant started by reading an use case description for 27 seconds, moving then to a sequence diagram for 48 seconds, and finally access the source code for 82 seconds.

[3] http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=EclipseHTMLEditor

[4] http://www.esurveyspro.com/

We pruned out from such logs browsing activities shorter than 5 seconds. Although this would remove some potentially useful information, we assume that such short activities are mainly due to the need for scrolling across various windows in the IDE.

5.2.5 Analysis Method

To answer RQ_1, we measure (in seconds) the time spent by participants on each of the artifact types considered in our study (i.e., the four different documentations plus source code). We also analyze the scores provided by the participants in the post-survey questionnaire to indicate their perceived usefulness of the exploited artifacts. Results are reported in terms of descriptive statistics and boxplots.

Besides analyzing the whole dataset collected during our study, we investigate whether participants with different levels of experience (graduate *vs.* undergraduate students) use artifacts differently. Due to the limited number of PhD students, and also for the sake of simplicity, we just distinguish between undergraduate (i.e., bachelor) and graduate (i.e., Master or PhD) students.

In addition to descriptive statistics and boxplots, we use Mann-Whitney test [81] to compare the proportion of time spent on each kind of diagram by participants having different levels of experience. We also evaluate the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [82] for ordinal data. We followed the guidelines in [82] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Still in the context of RQ_1, we verify if there is a correlation between the kind of artifacts exploited by participants and the correctness of the performed tasks. Note that our study does not aim at investigating whether the usage of different artifacts influences the task correctness. This cannot be done, because it would have required a specific controlled experiment with participants receiving different treatments, e.g., using some diagrams only, or "forced" to follow specific navigational paths only. Instead, this analysis should be considered as a form of sanity-check, to determine whether participants performed tasks seriously and whether participants using more specific kinds of artifacts could have suffered particular problems.

To measure the completeness and correctness of the tasks performed by each participant (i.e., her ability in correctly individuating the code components impacted by a maintenance activity), we used a combination of two well-known Information Retrieval (IR) metrics, recall and precision [80]. Recall measures the percentage of code components actually impacted by a maintenance activity correctly identified by a participant, while precision measures the percentage of identified components that are actually impacted. Since recall and precision measure two different (but related) concepts, we use their harmonic mean (i.e., F-measure

[80]) to obtain a balance between them when measuring task correctness.

The correlation between the type of artifacts exploited by participants and the correctness and completeness of the performed tasks is computed through (i) the Spearman correlation, performed between the time spent by participants in each task on each type of artifact and the correctness achieved in the task, and (ii) by building a logistic regression model for correctness based on the use (or not) of different kinds of artifacts.

Concerning RQ_2 we extracted, using the data derived by the FLOURITE plugin, information concerning how participants navigate different artifacts, and specifically:

- What artifacts did participant looked first, i.e., where the comprehension task started. Usually, one assumes this starts from requirements/use cases, although there are developers that start from source code directly.

- What artifacts did participants browse before getting to source code. This could potentially indicate the pattern followed to locate the source code element to be changed.

- What is the likelihood of making a transition from one kind of artifact to the other. This can likely indicate how the information gained by browsing a certain kind of artifact raises the need for accessing another kind of artifact, e.g., browsing source code after accessing sequence or class diagrams, or else looking at static models after dynamic models.

- What are the most frequently followed patterns. This was done by matching regular expressions of length varying from two to four onto the mined logs, and determining for each pattern whether it was iterated, e.g., participants could go back and forth between source code and class or sequence diagrams repeatedly.

Finally, we investigated whether participants with different levels of experience followed different patterns and whether following certain patterns can influence the task correctness.

All statistical analyses of this study have been performed using the *R* environment [74]. For all statistical procedures, we assumed a significance level of 95%.

Replication package To facilitate the replication of this study, a complete replication package is available[5]. It includes (i) an Eclipse installation bundle, with all the exploited plug-ins installed and the object system SMOS (source code and other artifacts) already imported, (ii) the task description for all 8 tasks, (iii) the post-study questionnaire, and (iv) the FLUORITE logs for the 33 participants. Also, the package includes the working data set with our study results.

[5] http://distat.unimol.it/reports/icsm-docs/

Table 5.1: Recall, Precision, and F-measure achieved by participants when performing the tasks.

Dataset		Recall	Precision	F-measure
Undergraduates	Mean	0.65	0.79	0.71
	Median	0.81	1.00	0.82
	St. Dev.	0.40	0.38	0.37
Graduates	Mean	0.67	0.88	0.76
	Median	0.88	1.00	0.93
	St. Dev.	0.37	0.31	0.35
All	Mean	0.67	0.85	0.75
	Median	0.88	1.00	0.86
	St. Dev.	0.38	0.34	0.36

Table 5.2: Use (percentage of tasks and time spent) of different kinds of artifacts: descriptive statistics.

Artifacts	Tasks (%) (All)	Tasks (%) Undergrad.	Tasks (%) Graduate	Time spent (all data, %)			
				mean	1Q	median	3Q
Use case	33	28	36	3	0	0	2
Sequence Diagram	72	68	74	10	0	7	16
Class Diagram	60	49	66	13	0	4	15
Javadoc	15	21	11	2	0	0	0
Source Code	100	100	100	72	66	79	89

5.3 Analysis of the Results

Before answering the research questions formulated in Section 5.2.2, it is important to verify whether participants seriously performed the assigned tasks. To this aim, Table 5.1 reports the average values for recall, precision, and F-measure achieved by undergraduate and graduate students, as well as when considering the entire dataset. Results show that participants were able to achieve quite good performances, with an average F-measure of 0.75. This sanity check makes us confident that participants seriously performed the assigned tasks. Also, as expected, graduate students achieved, on average, better performances than undergraduate students (+5% in terms of F-measure).

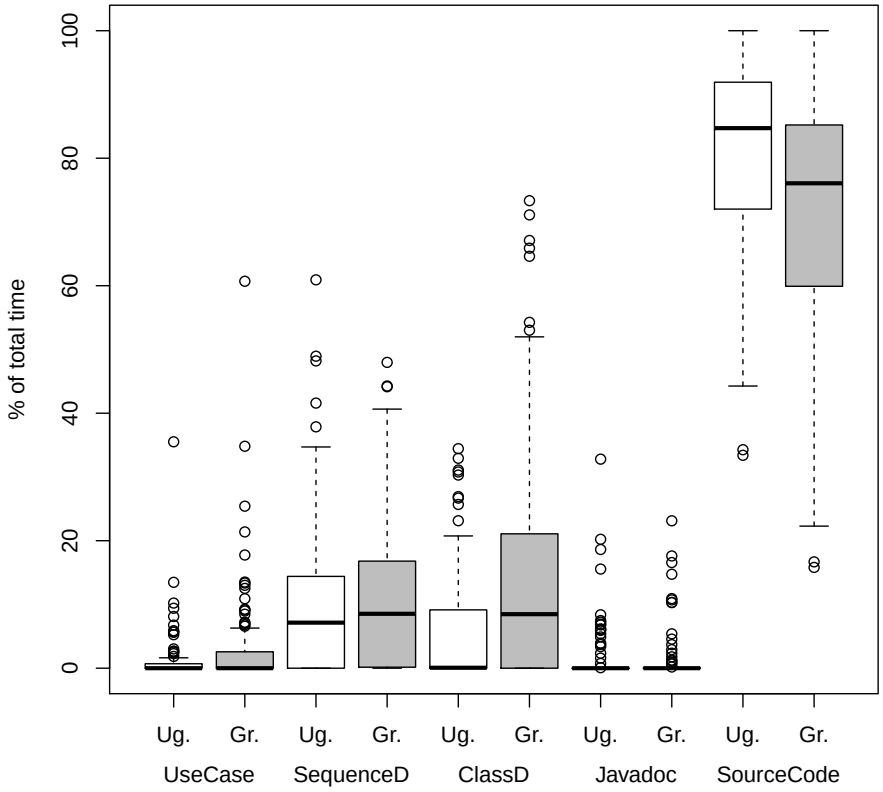


Figure 5.2: Usage (in percentage) of different kinds of artifacts. Ug = undergraduate students, Gr = graduate students.

5.3.1 RQ_1: How much time did participants spend on different kinds of artifacts?

Table 5.2 reports the percentage of tasks in which each kind or artifact has been used (for the entire dataset as well as by separately considering participants with different levels of experience), and descriptive statistics about the percentage of time spent on various kinds of artifacts (by considering the entire dataset). Fig. 5.2 shows boxplots of such percentage for different levels of experience.

If considering the whole dataset, and analyze the time spent on artifacts (right-side of Table 5.2), results indicate that participants spent most of their time (72% on average) on

source code. This result is mainly due to two reasons. First, even when participants were able to identify the impacted components by analyzing documentation artifacts we observed that they checked-back in the source code that the identified methods/attributes were actually there and really impacted by the maintenance activity to perform. This suggests a kind of distrust with respect to documentation artifacts, as also confirmed by the fact that source code has been used in 100% of the tasks. Second, source code clearly requires more time to be read and understood as compared to the artifacts present in the documentation. In particular, participants spent, on average, 154 seconds on each source code file, compared to the 70 spent on a class diagram, 49 on a Javadoc file, 35 on a sequence diagram, and 34 on a use case.

If we look at the percentage of tasks in which each kind of artifact was used at least once (left-side of Table 5.2), we notice that—besides source code, obviously used in 100% of the tasks—the most commonly used documentation artifacts are class and sequence diagrams. The latter were used in 72% of the task. On such diagrams, participants spent on average 10% of their time (median=7%). Only one of the 33 participants did not exploit at all sequence diagrams during the tasks and justified such a choice in the post-study questionnaire: *"sequence diagrams would be useful only if class diagrams were not present"*. However, as we will see shortly, this is an isolate point-of-view.

As for class diagrams, they were used in 60% of tasks and participants spent, on average, 15% of their time on them (median=4%). This strong misalignment between the mean and the median values for class diagrams highlights that, while generally they are used for a lower proportion of time as compared with sequence diagrams, some participants spent a very high proportion of their time on class diagrams, as also shown by the outliers reported in Fig. 5.2. Two participants did not use at all class diagrams in the tasks.

Turning to use cases, they were used in 33% of tasks by participants, which focused on them just the 3% of their time, on average. As said before, participants spent just 34 seconds, on average, on each consulted use case against, for instance, the 154 spent on each source code file. Among the 33 participants, three of them did not access at all use cases.

Finally, Javadoc documentation was not used a lot by participants of our study. They accessed Javadoc in just 15% of the tasks. Also, 11 participants out of 33 never open Javadoc files during the tasks.

Concerning the time spent by participants with different experience levels on different artifacts, Fig. 5.2 and the results of the Mann-Whitney test reported in Table 5.3 indicate that: (i) there is no significant difference in accessing use cases and sequence diagrams; (ii) graduate students use class diagrams significantly more than undergraduates, with a medium effect size; (ii) undergraduates students used source code and Javadoc significantly more than

Table 5.3: Percentage of time spent on artifacts by participants with different experience: Mann-Whitney test and Cliff's d effect size (positive values indicate differences in favor of graduate students, negative in favor of undergraduates).

Artifact	p-value	Cliff's d
Use Case	0.1020	0.1030
Sequence Diagram	0.3102	0.0749
Class Diagram	**0.0001**	0.2757
Javadoc	0.0268	-0.1040
Source Code	$<$ **0.0001**	-0.2939

graduate students, with a small and medium effect size respectively. Such results partially contradict those of other studies [125], which indicated that junior developers tend to benefit of models than senior developers, that tend to directly focus onto source code.

To better understand the results of the quantitative analysis, we analyzed the feedbacks provided us by means of the post-study questionnaire. Fig. 5.3 shows boxplots—for different levels of experience—of the ratings provided by participants to the usefulness of the different kinds of artifacts used. As explained in Section 5.2.3, one corresponds to classify a kind of artifact (documentation as well as source code) as "totally useless", while five indicates a "very useful" kind of artifact.

As we can notice, sequence diagrams are considered to be the most useful kinds of artifact, with a mean score of 4.3 for both undergraduates and graduates (median 4 for undergraduates and 5 for graduates). Some of the comments left by participants in the post-study questionnaire explain the reasons behind this evaluation. Several of them explained how *"once found the sequence diagram(s) describing the feature(s) involved in a change request, it was easy to identify the candidate impacted components. This strongly speeds up the tasks."* Others explained as sequence diagrams *"represent a fair compromise between use cases (too abstracts) and class diagrams (providing useless details about an entire subsystem)"*.

Class diagrams and source code were generally ranked as equally useful. However, while undergraduates found source code slightly more useful (mean 3.6, median 4) than graduates (mean 3.2, median 3.5), the opposite happens for class diagrams, that were found more useful by graduate students (mean 3.9, median 4) than by undergraduates (mean 3.4, median 3). Among the 8 participants that considered class diagrams very useful, five of them explained as *"it is easy to map class diagrams on source code, and thus to fast check the candidate impacted components identified from the diagram."* Five of the 33 participants declared the source code as the most useful artifact. The perceived reason is that: *"while the provided high-level documentation is useful to speed-up the task, consulting source code is mandatory to perform some of them, like the required bug-fixes."*

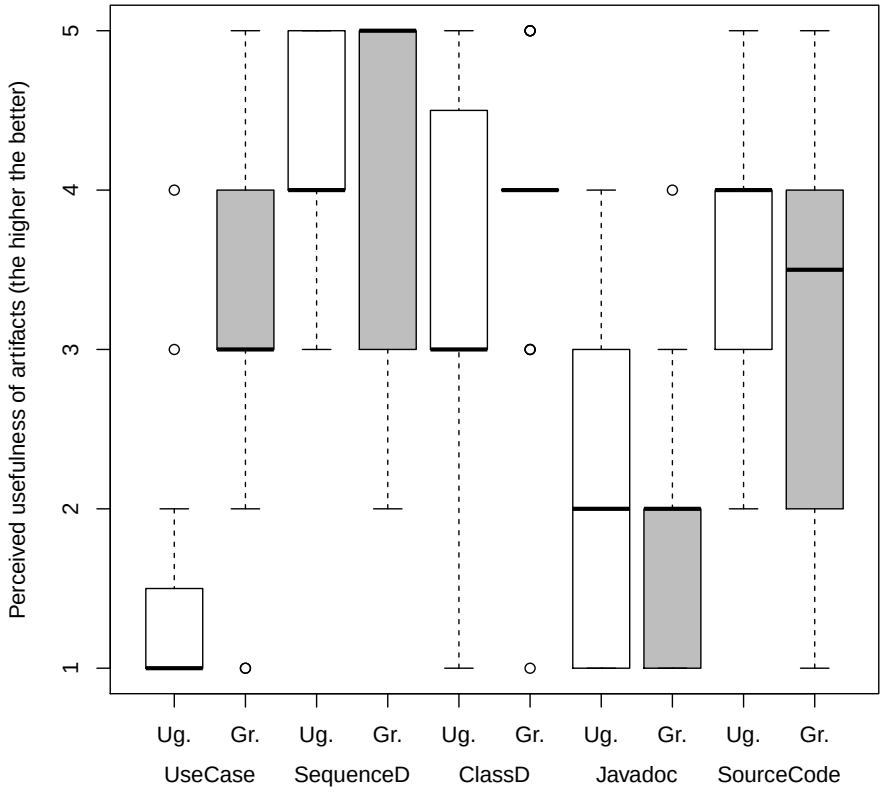


Figure 5.3: Perceived usefulness of the different kinds of artifacts as indicated by participants. Ug = undergraduate students, Gr = graduate students.

As for use cases, it is interesting to note that the usefulness assessment provided by graduates (mean 3.2, median 3) is higher than for bachelor (mean 1.5, median 1). This is the only case for which the Mann-Whitney test reveals a statistically significant difference (p-value=0.002, Cliff's d 0.68 – high), while for all other artifacts the differences between the two levels of experience are not significant. This suggests how more experienced participants are able to start the task from requirements/use cases before accessing models and source code. Undergraduates failed to explain use cases, as they tried to identify object names within them *"in use cases it was not possible to find information about components of the system impacted by a change"*, rather than relying on use cases to identify the piece of functionality to be changed before accessing sequence/class diagrams.

Finally, the provided feedbacks confirmed that Javadocs were perceived as the least useful artifacts. For Javadoc, the mean and median score was 2 ("useless") for both undergraduate and graduate students. Participants declared that *"with the other sources of documentation available Javadoc became useless to identify impacted components."* This is to say, our study does not show that Javadoc is useless: it is likely to be very useful during development activities, e.g., when using a new API. Instead, it provides a limited (or no) support when analyzing the impact of a change.

As explained in Section 5.2.5, we also analyzed the presence of possible correlations between the time spent by participants on the different kinds of artifacts and the correctness of the performed tasks in terms of recall, precision, and F-measure. By applying the Spearman correlation test no interesting correlations were found for undergraduate and graduate students, as well as when considering all participants as a single dataset. Also, a logistic regression model for correctness based on the use (or not) of different kinds of artifacts did not lead to any significant result, i.e., none of the artifacts resulted significant in the model.

RQ_1 summary:

[1] Participants spent more time to analyze Low-Level Artifacts as compared to High-Level Artifacts.

[2] Participants consider sequence diagrams as the most useful source of documentation when performing the required tasks, followed by class diagrams and source code.

[3] Undergraduate students spent a significantly higher proportion of time on source code than graduate students that, instead, spent more time on class diagrams.

5.3.2 RQ_2: How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?

Table 5.4 reports, for each kind of artifact used in our study, the number and percentage of tasks participants started from such artifact. The most frequent starting point is by far source code (42% of the tasks), followed by sequence diagrams (25%), class diagrams (17%), use cases (12%), and Javadoc (3%). Note that this result is quite surprising since one could expect that developers start their analysis from HLA going down to the code. Instead, in our study 84% of the tasks started from source code and design models, i.e., class or sequence diagrams.

Table 5.4: What participants looked first.

Artifact	All data		Undergrad.		Graduates	
	#of Tasks	Perc (%)	# of Tasks	(%)	# of Tasks	(%)
Use Case	31	11.92	3	3.16	28	16.97
Sequence Diagram	66	25.38	23	24.21	43	26.06
Class Diagram	45	17.31	9	9.47	36	21.81
Javadoc	9	3.46	4	4.21	5	3.03
Source Code	109	41.92	56	58.95	53	32.12

When observing data for different levels of experience (right-side of the table), what we notice is pretty consistent with findings of $\mathbf{RQ}_1$ concerning the proportion of usage for different kinds of diagrams. Basically, undergraduates tend to start tasks mainly using source code (58%), while this percentage is only 32% for graduates. The percentage of participants starting with sequence diagrams is similar (24% for undergraduates, 26% for graduates), while graduates tend to start with class diagrams more than undergraduates (22% vs 9%). Finally, there is a non-negligible proportion of graduates that starts from use-cases (17%, vs. 3% of undergraduates). This is likely due to the fact that graduate students have a better training on software engineering principles and on how using models and HLA during maintenance tasks, and also because they have more experience in evolving existing systems.

Since we found that in 58% of cases source code does not represent the entry point, we analyzed what are, in these cases, the pattern followed by participants before reaching source code. Table 5.5 reports them (using through regular expressions). In a similar proportion of tasks, participants access sequence or class diagrams before going to source code. This happens in 71 tasks, 36 for sequence (14%) and 35 for class (13%) diagrams.

Another frequently followed path consists of one or more switches between sequence and class diagrams. This path is more frequent starting from the sequence (22 tasks)—row (SD)+ in Table 5.5—than from class diagrams (7 tasks)—row (DS)+ in Table 5.5. In both cases, participants tried to gain source code knowledge from its most direct model representations (i.e., class and sequence diagrams) before going through it. Also, for 18 tasks, participants switch one or more times between use case (used as starting point) and sequence diagrams—row (US)+ in Table 5.5. Overall, it is interesting to note that sequence diagrams are accessed in four out of the five most frequent path followed before reaching source code. Other paths reported in Table 5.5 are quite uncommon, e.g., opening a use case (row U) or a Javadoc file (row J).

When looking at results by different levels of experience (right-side of Table 5.5), it can be noticed that, besides what it is known already from previous analyses, graduate students use

Table 5.5: Patterns followed before reaching source code.

Pattern	All data		Undergrad.		Graduates	
	# of Tasks	(%)	# of Tasks	(%)	# of Tasks	(%)
S	36	13.85	17	17.89	19	11.51
D	35	13.46	8	8.42	27	16.36
(SD)+	22	8.46	2	2.10	20	12.12
(US)+	18	6.92	2	2.10	16	9.70
U(SD)+	7	3.46	1	1.05	6	3.64
(DS)+	7	2.69	1	1.05	6	3.64
J	4	1.54	3	3.16	1	0.60
U	4	1.54	0	0.00	4	2.42
S(US)+	3	1.15	1	1.05	2	1.21
SU(SD)+	2	0.77	0	0.00	2	1.21
Other	13	5.00	4	4.21	9	5.45

S = Sequence Diagram, D = Class Diagram
U = Use Case, J = Javadoc

Table 5.6: Average transition frequencies between the kinds of artifacts.

From/To	U	S	D	J	C
U		56%	8%	0%	36%
S	5%		17%	1%	77%
D	2%	18%		2%	78%
J	0%	6%	16%		78%
C	7%	49%	37%	7%	

S = Sequence Diagram, D = Class Diagram
U = Use Case, J = Javadoc, C = Source Code

much more navigation patterns across different kinds of diagrams. As the table shows, undergraduates just looked at sequence or class diagrams before diving into source code. Instead, graduate students also followed more complex navigation patterns, e.g., sequence+class (with some iterations), use case+sequence (with some iteration), or even use cases followed by iterations on sequence and class diagrams. Once again, this indicated that people with more experience are more prone to follow an "integrated" approach when performing a comprehension task.

Then, we analyzed the transition frequencies between the different kinds of artifacts used

in our study. Table 5.6 reports the results considering the entire dataset. As it can be noticed, the most frequent transitions are toward the source code (column C), 77% of which are from a sequence diagram, and 78% from class diagrams and Javadoc files. The take-away of these results is that, after have gathered information from one of those kinds of artifacts, developers try to map them into source code elements. Note that this is true also when separately analyzing participants having different experience levels with small changes in the transition frequencies.

The behavior of participants when reading use cases is, instead, pretty different from the one observed above. They shift toward source code in just 36% of times, privileging the reading of a design diagram (64% of the cases, 56% for sequence and 8% for class diagrams) before reaching source code. However, when analyzing the data for participants having different experience, some differences came out. In particular, graduate students tend to consult a low-level diagrams after accessing an use case (72%, 64% for sequence and 8% for class diagrams), against the 43% of undergraduates (35% for sequence and 8% for class diagrams). After reading an use case, undergraduates go to source code in 56% of cases, against the 27% of graduates. This further confirms that more experienced developers are more prone to use different sources of documentation when performing a comprehension task.

Other common transitions between different kinds of artifacts occur (i) when reading a sequence diagram toward a class diagram (17%) and (ii) when reading a class diagram toward a sequence diagram (18%). In this case, no interesting difference has been observed between participants having different experience.

It is also interesting to analyze what other artifacts participants access immediately after browsing source code. Table 5.6 indicates that participants go back from source code to documentation just to access design diagram, i.e., sequence (49%) and class (37%) diagrams. Again, no important differences were found between participants having different experience.

Finally, we analyzed the most frequent navigational patterns followed by participants during the tasks. Table 5.7 and Fig. 5.4 report information about the six most frequent patterns we found. In particular, Table 5.7 reports the number and percentage of occurrences on the whole dataset and for participants with a different degree of experience, whereas Fig. 5.4 shows the boxplots for the distribution of its repetitions (i.e., the number of times a pattern appears in a single task). As it can be expected according to what observed so far, the most frequent pattern consists of going back and forth from sequence diagram to source code: this occurred in 153 tasks (59%). The median of its repetitions is two, but we also found cases where this pattern has been repeated more than 10 times in a single task. Another very frequent pattern is that going back and forth from class diagrams to source code, present in 128 tasks (49%) with also a median repetition of two. Among the longer patterns (i.e., those

Table 5.7: Most frequent navigational patterns.

Pattern	All data		Undergrad.		Graduates	
	Occ.	(%)	Occ.	(%)	Occ.	(%)
USDC	12	4.62	0	0.00	12	7.27
USD	21	8.08	2	2.11	19	11.52
USC	35	13.46	10	10.53	25	15.15
UDC	13	5.00	3	3.16	10	6.06
SDC	55	21.15	15	15.79	40	24.24
SC	153	58.85	58	61.05	95	57.58
DC	128	49.23	39	41.05	89	53.94

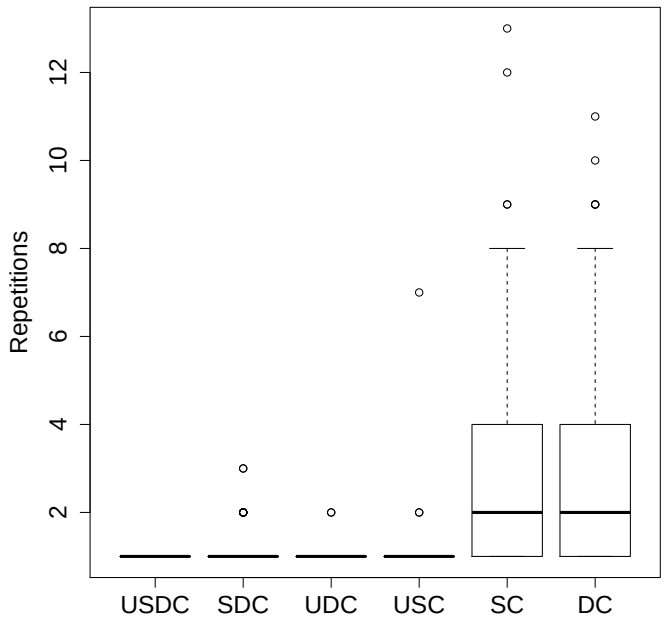


Figure 5.4: Most frequent navigational patterns and distribution of their repetitions. S = Sequence Diagram, D = Class Diagram, U = Use Case, C = Source Code.

having a length > 2), the most frequent is that going from sequence to class diagram and then to source code (SDC in Fig. 5.4). This pattern has been followed by participants in 21% of the performed tasks, generally with a single repetition. Also, in 13% of tasks, participants

went from use cases toward sequence diagrams, and finally to the code. In addition, from the analysis of Fig. 5.4 we can conclude that (i) Javadoc is not present in any of the most common patterns; and (ii) all common patterns end (as expected) with a source code artifact.

When looking at the occurrences of patterns among participants with a different level of experience (right-side of Table 5.7), we can notice that (i) the SC pattern (sequence+code) is consistently followed by about 60% of both undergraduates and graduates; (ii) patterns involving use cases (USDC, USD, USC, and UDC) are much more frequent for graduate than for undergraduates; and (iii) for what concerns longer patterns followed by undergraduates, the SDC pattern was followed in 16% of the cases, and USC in 10% of the cases. In summary, we can notice a higher proportion of patterns reflecting a more "integrated" approach for graduates. Also, graduate students followed patterns involving class diagrams and code (DC) more (54%) than undergraduates (41%). We did not notice any significant difference in the number of iterations for all the above mentioned patterns, except for the SC pattern, that received a median of 3 iterations for undergraduates, that used it and of 2 iterations by graduates that used it. The difference is statistically significant (p-value =00017) and the Cliff's d effect size medium ($d = 0.293$). In other words, less-experienced participants had to go back and forth between sequence diagrams and source code more than experienced ones to locate the methods to be changed. As done for RQ_1, we also statistically verified the relationship between the patterns followed by participants and the correctness of their tasks. In particular, we built a logistic regression model for correctness with respect to the use (or not) of the different patterns. Also in this case, we did not find any statistically significant result.

RQ_2 summary:

[1] Participants tend to start the assigned task from source code or from design documents, i.e., class and sequence diagrams.

[2] More experienced participants tend to follow a more integrated approach than less experienced ones, traversing different kinds of diagrams, e.g., starting from use cases, and then browsing design documents, until reaching source code.

[3] During their task, participants tend to go back and forth repeatedly between source code and to design diagrams (sequence and class diagrams).

5.4 Threats to Validity

This section discusses the threats that could affect our results.

Threats to *construct validity* concern the relation between the theory and the observation. In our study, this threat can mainly be due to errors in the collected measurements. For what concerns capturing participant's browsing activities, we relied on an existing tool (FLUORITE), making sure each participant had correctly installed it, and carefully instructed them how to browse artifacts in Eclipse while using the tool. When collecting results, we discarded cases of short access to artifacts (less than five seconds) that are unlikely to be an indication of reading the document, but rather of scrolling different documents. Clearly, this might have meant loosing some quick, but valid, accesses.

Another threat concerns the way the correctness of the task is evaluated, i.e., by means of precision, recall, and F-measure computed over the list of elements to be modified as identified by participants. On the one hand this allows a subjective evaluation and allows to perform a comprehension task without requiring the execution of source code. On the other hand, this can provide a coarse-grained and partial evaluation of how the comprehension task was performed.

Threats to *internal validity* concern any confounding factor that could influence our results. For example, such a threat may be due to the fact that some participants might have decided not to browse diagrams because they were unreadable or the tool was not usable. To mitigate such a threat, we avoided to use any specific UML modeler (we used PDF documents instead), and we produced diagrams large enough to be easily readable.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. As explained in Section 5.2, this is more an observational study rather than a controlled experiment, as all participants received the same treatment. Wherever possible, however, we used appropriate statistical procedures and effect size measure to support our claims.

Threats to *external validity* concern the generalization of our findings. This study has been conducted with students, and for this reason the obtained results may not generalize to professionals, which might be used to perform comprehension task using high-level artifacts in a different way, or in some cases not using them at all. To some extent, our participants can be considered as representative of junior developers, joining a project as newcomers to perform a maintenance task.

5.5 Related work

Several studies have been performed to analyze the benefits of UML documentation during software development and evolution [126]. In the next section we focus the attention on studies analyzing the effect of documentation on maintenance/comprehension tasks. In addition, we also discuss study carried out to analyze the behavior—from different perspectives—of developers performing maintenance tasks.

5.5.1 Impact of UML documentation on Maintenance Tasks

Experiments aimed at studying the impact of UML documentation in software maintenance [116] indicated that such a documentation improves the functional correctness of changes and the quality of the design. While simple class diagrams, with or without stereotypes, help low ability or low experience participants, a complete, thorough UML documentation requires a certain learning curve to become useful [116]. In fact, in some cases the previous experience of participants influences the understandability of UML diagrams. Torchiano [127] showed that object diagrams have a significant impact on comprehension tasks, when compared with UML documentation consisting of class diagrams only.

Dzidek *et al.* [117] performed a controlled experiment aimed at investigating the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of software systems. In the context of the experiment, participants (represented by professional developers) performed evolutionary tasks with and without UML documentation. Their results indicated that participants using UML documentation were able to statistically increase the correctness of changes. Also, they were able to slightly improve the design quality at the expense of an insignificant increase in development time caused by the overhead of updating the UML documentation.

UML limitations in aiding program understanding are highlighted in experiments performed by Tilley and Huang [121]. They highlighted that UML does not provide a sufficient support to represent domain knowledge. Lemus *et al.* [119] showed that composite states improve the understandability of statecharts provided that participants had a previous experience in using them. This also happens when UML is complemented with complex formalisms, such as the Object Constraint Language (OCL) [128]: a substantial training is required to make OCL useful, although for some tasks OCL better helped low ability participants, who were not able to guess system functionality from the textual description.

The role of dynamic UML diagrams in software comprehension was investigated by Otero and Dolado [120]. The comprehension level and the time required to perform the comprehension task resulted different for different diagrams and system complexities. Abrahão *et*

al. [129] also analyzed the support given by sequence diagrams during the comprehension of functional requirements. The results showed that sequence diagrams improve the comprehension of the functional requirements in the case of high ability and more experienced participants.

We share with the aforementioned studies the need to analyze the support given by software documentation during software evolution. However, we did not focus on a specific kind of documentation. Instead, we provided to participants several documentation artifacts aim at studying which are the most used artifacts and how developers use such artifacts.

Tryggeseth [118] conducted a study, for some aspects similar to our, to analyze the impact of the availability of up-to-date documentation on maintenance tasks. In the context of the experiment participants were asked to perform maintenance tasks with and without software documentation (represented by requirements specification, design document, test report, and user manual). Their results indicated that participants using the available documentation spent less time to understand how to implement a change request. Besides reducing the time, the documentation also allowed participants to better understand the system and provided more detailed solution on how to incorporate the changes.

While we share with Tryggeseth the need for analyzing the impact of several software documentation artifacts on maintenance tasks, our study presents two main differences: (i) we analyzed how developers use documentation during software evolution aimed at identifying particular navigation paths; and (ii) we also investigated the effect of experience on how participants follow different usage paths.

5.5.2 Studies about Developers' Behavior during Maintenance Tasks

von Mayrhauser and Vans [115] observed how professional developers work when performing maintenance tasks, finding that programmers use a multi-level approach during source code understanding, switching between different programs as well as between different sources of documentation.

Robillard *et al.* [130] performed an exploratory study to analyze the factors that contribute to effective program investigation behavior, while Sillito *et al.* [131] performed two qualitative studies aimed at understanding what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools help in that process.

Singer *et al.* [132] studied the daily activities of developers. Such a study provides some guidelines fro tool designers that represent an alternative to the traditional paths taken in human-computer interaction, namely those issuing from the study of the users' cognitive processes and mental models, and the emphasis on usability. Also, DeLine *et al.* [133] iden-

tified several usability issues of conventional development environments when a developer has to update a software system, including maintaining the number and layout of open text documents and relying heavily on textual search for navigation.

de Alwis and Murphy [122] analyzed how programmers experience disorientation when using Eclipse, identifying three factors that may lead to disorientation: the absence of connecting navigation context during program exploration, thrashing between displays to view necessary pieces of code, and the pursuit of sometimes unrelated subtasks.

Storey *et al.* [134] performed a study aimed at analyzing whether program understanding tools enhance or change the way that programmers understand programs. Based on the results achieved the authors suggested that tools should support multiple strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration.

The behavior of software developers has also been analyzed aimed at identifying approaches able to reduce the information overload (e.g., number of artifacts to be analyzed) of developers by filtering and ranking the information presented by the development environment [123, 135, 136]. The findings of our study can complement such models. The usage patterns identified in our study can be used to complement such approaches providing a more effective support during program comprehension.

Recently, eye tracking systems have been used to investigate the comprehension of UML diagrams [137, 138], the effect of the layout on the comprehensibility of software documentation artifacts [139], and the effect of design patterns on comprehension [140]. The use of eye tracking systems is particular useful to investigate on the way developers look at the documentation aimed at deriving guidelines for facilitating the comprehension of software documentation.

We share with all these studies the need to empirically analyze the behavior of developers during software development and maintenance. However, we analyzed the behavior from a different perspective. Specifically, our analysis aimed at analyzing how developers use software documentation in order to identify recurring usage paths. Such paths could be used to enhance contemporary IDEs and provide more effective strategies for browsing documentation artifacts.

5.6 Summary

This study reported a study aimed at investigating how developers navigate and browse documentation artifacts during maintenance tasks. We asked 33 participants to perform 8 different maintenance tasks on a Java software system providing them, besides the source code, use

case descriptions, sequence diagrams, class diagrams, and Javadocs. Through an Eclipse plugin, we recorded how much time participants spent on different artifacts, and how they navigated from an artifact to another.

Results of our study indicated that participants spent most of their time on source code when identifying code components impacted by a maintenance activity, while preferring sequence diagrams among the available sources of documentation, followed by class diagrams. Also, they generally started their tasks from source code, or from design documents (84% of cases), then browsing back and forth between source code and either class or sequence diagrams. Less frequently, participants—especially more experienced ones (i.e., graduate students)—-followed an "integrated" approach, by using different kinds of artifacts, namely starting from use cases, then accessing design documents (class and/or sequence diagrams), and finally accessing source code. This analysis can be useful to build recommenders in help newcomer to choice appropriate patterns in navigate software documentation when apply maintenance tasks.

Chapter 6

Labeling Source Code with Information Retrieval Methods

Contents

In the previous Chapter we discussed the results of a study—conducted with 33 participants—aimed at investigating (i) to what extent newcomers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts. Results indicate that, although newcomers spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, developers—especially more experienced ones—follow an "integrated" approach by using different kinds of artifacts. Such information can be seen as a starting point to built recommenders in help newcomer to choice appropriate patterns in navigate software documentation when apply maintenance tasks. However, sometime the code complexity represents a barrier that slow down the program comprehension, as well as, the possibility of a newcomer to become active and terms of code changes. What a newcomer needs in such situation is a summary of code that help him/her in comprehension of complex code. This Chapter reports an empirical study aimed at investigating to what extent a source code labeling based on IR techniques would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task. In recent years, researchers have applied various IR methods to "label" software artifacts by means of some representative words. Such "summaries" can help to improve the program comprehension of project newcomers that are join a software project. Thus, the key goal of this chapter try to generate high quality source code summaries, useful for projects newcomers in understanding source code elements. Results show that overall there is a relatively high overlap between automatic and human-generated labels, ranging between 50% and 90%. However, the highest overlap is obtained by using the simplest heuristic, while the most sophisticated techniques, i.e., LSI and LDA, provide generally the worst accuracy. One reason of the result is that developers mainly used words from class names, method names and signatures, and (partially) from class and method comments to label artifacts. We also found that the high entropy of terms in the classes inhibits the capability of topic modeling techniques—i.e., LSI and LDA—to efficiently identify and cluster topics in source code. This result highlights that approaches such as LDA and LSI are worthwhile of being used when analyzing heterogeneous collections, where documents can contain information about multiple topics [141]. Unfortunately, such an heterogeneity is not always present in source code artifacts. Thus, the *ad-hoc* heuristics experimented in this study represent a valid approach to *build high quality summaries* of source code elements to help new developers in program comprehension.

6.1 Motivation: support program comprehension with source code summaries

Program comprehension is a key activity for software maintenance and evolution. The importance of program comprehension has been summarized by Rajlich and Wilde: "*software that is not comprehended cannot be changed*" [142]. During program comprehension, developers read the source code aiming at building a *cognitive model* that is used to form a *mental model*, i.e., the developers' mental representation of the program to be understood and changed [143].

Such a cognitive process could be tedious, error prone and time consuming in large software systems, where especially for junior developers of a project is requested to read (and comprehend) a large number of source code lines. In such a scenario, developers spend more time reading and navigating the code than writing it [144, 145]. Recently, several approaches have been proposed to facilitate the comprehension of large software systems. The key idea is to reduce the amount of information to read and comprehend by providing a "short" description of the source code which can be read quickly. In order to reach such a goal, the source code is automatically labelled by means of some representative words.

IR techniques are used to identify—in source code identifiers and comments—keywords that properly describe the artifact. Such a representation provides a bird-eye's view of the source code artifacts, that allows developers to look over software components quickly, and make more informed decisions on which parts of the source code they need to analyze in detail [146]. Indeed, many researchers have applied—in recent and past years—IR techniques to automatically "label" software artifacts. For example, Kuhn *et al.* [147] used discriminant words from LSI concepts to label software packages; Thomas *et al.* [148] used LDA to label source code changes; Gethers *et al.* [149] used Relational Topics Model (RTM) to identify and relate topics in HLA and source code.

Despite the aforementioned research efforts, up to now few studies have been performed to analyze whether these automatic techniques are able to extract words that make sense and are actually relevant to developers [146, 150]. This lack motivates our work. Specifically, we investigate to what extent an IR-based source code labeling technique is able to identify relevant words in the source code, compared to the words humans would manually select during a program comprehension task. In essence, we aim at verifying whether the terms identified by an automatic technique overlap with terms selected by developers when building their mental model of a source code component. Specifically, we are interested to identify relevant words for newcomers that are joining a software project.

To this aim we conducted two experiments in which we asked 17 Bachelor's Students

and 21 Master's Students, respectively, to describe 20 Java classes taken from a Java software system—JHotDraw[1]—using at most ten words extracted from the class source code and comments. Then, we analyzed:

[1] to what extent the keywords identified using various IR techniques, i.e., VSM, LSI, LDA, and some *ad hoc* heuristic picking terms from specific part of the source code and comments, overlap with those identified by humans;

[2] what kind of source code (and comment) elements were used by subjects to produce the labels; and

[3] what characteristics of the analyzed artifacts could influence the effectiveness of the various techniques used to automatically produce labels.

Results show that, overall, automatic labeling techniques are able to well-characterize a source code class, as they exhibit a relatively good overlap—ranging between 50% and 90%—with the manually-generated labels. The highest overlap is obtained by using the heuristic considering only terms extracted from method signatures and class comment. LSI and LDA (which are based on artifact clustering) generally provide the worst overlap. In particular, we observed that the high entropy of terms contained in the source code artifacts—i.e., artifacts do not contain terms that dominate over the others—inhibits the capability of such techniques to efficiently identify and cluster topics in source code. Indeed, LSI and LDA were designed for analyzing heterogeneous collections, where documents can contain information about multiple topics [141]. Unfortunately, this heterogeneity is not always present in source code artifacts, especially when considering a single class having a well-defined set of responsibilities, and thus few and strongly-coupled topics. In such a scenario, it is difficult to identify dominant terms that could be used to characterize a class in terms of topics.

We also analyze the effect of other factors (e.g., comment verbosity) on time required to participants for Labeling Source Code (**LSC!**) as well as the effectiveness of automatic source code labeling techniques. We observed that the higher the comment verbosity, the lower the time required by subjects to identify the keywords. This result confirms the importance of having comments in comprehension tasks [151, 152]. In addition, we also observed that clustering-based approaches (i.e., LSI and LDA) are worthwhile to be used on source code artifacts having a high comment verbosity, and also on artifacts requiring more effort to be manually labeled.

This Section is organized as follows. Section 6.2 describes the empirical study definition and planning. Section 6.3 reports and discusses the results. Section 6.4 discusses the threats

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

that could affect the validity of our study. Section 6.6 summaries the results of our empirical study.

6.2 Study Definition and Planning

In the following, we report the definition and planning of our empirical study. The experiment replication package and working data sets of its results are available online[2]. Is important to note that in the following section we report the results obtained for only one software project (for reason of space), but more details about the complete study conducted is available in our bibliography in [3]

6.2.1 Study Definition

The *goal* of our study is to create a precise source code labelings automatically generated by means of IR techniques and/or other simple heuristics. The *quality focus* concerns the quality of automatically-generated source code labels, measured as their overlap with the human-generated labels. The *perspective* is of researchers interested in understanding to what extent automatic source code labeling approaches based on IR methods or simple heuristics can be used, and in which circumstances each technique performs well or not.

6.2.2 Study Context

The *context* of our study consists of *objects*, i.e., classes extracted from two Java software systems, and *subjects*, i.e., undergraduate students from the University of Molise, Italy, and graduate students from the University of Salerno, Italy.

Specifically, the object systems used in our study are JHotDraw and eXVantage. JHotDraw[3] is an open source vectorial drawing tool, developed with the purpose of illustrating the usage of design patterns. In our study we used version 6.0 b1 of JHotDraw, which con-

[2]http://distat.unimol.it/reports/labeling/

[3]http://www.jhotdraw.org

Table 6.1: Classes from JHotDraw and eXVantage used as objects of our study

JHotDraw	eXVantage
org.jhotdraw.draw.GraphicalCompositeFigure	com.avaya.exvantage.decision.gui.Browser
org.jhotdraw.draw.TextTool	com.avaya.exvantage.structures.cfg.cfgmanager.CFGManager
org.jhotdraw.draw.SelectionTool	com.avaya.exvantage.source.representation.ast.CodeFragment
org.jhotdraw.io.ExtensionFileFilter	com.avaya.exvantage.structures.dependency.DependencyGraph
org.jhotdraw.app.action.OpenAction	com.avaya.exvantage.util.graph.EdgeElement
org.jhotdraw.draw.GroupFigure	trace.EventThread
org.jhotdraw.util.prefs.PreferencesUtil	com.avaya.exvantage.ui.interfaces.cli.ExvantageCommand
org.jhotdraw.draw.ArrowTip	com.avaya.exvantage.trace.format.session.TraceBitEncoder
org.jhotdraw.draw.GridConstrainer	com.avaya.exvantage.ui.trace.tracetransfer.TraceTransfer
org.jhotdraw.draw.TextAreaTool	com.avaya.exvantage.decision.util.graph.VarMatcher

sists of 275 classes (29 KLOC). eXVantage[4] is a novel testing and test data generation tool developed in an industrial environment. The version used in our study (V20090507173755) is composed of 348 classes (28 KLOC). In the context of our study, we selected ten classes from JHotDraw and ten from eXVantage. We selected classes that are not too trivial, nor too complicated to be understood by the experiment participants. Table 6.1 reports the list of classes we selected from the systems.

Concerning the subjects involved in our study, we performed two experiments. The first one—in the following referred as *Experiment 1*— involved 17 Bachelor's students attending the Software Engineering (SE) course at the University of Molise, Italy. The second one—in the following referred as *Experiment 2*—involved 21 Master's students attending the course of Advanced Software Engineering at the University of Salerno, Italy. In both cases, all subjects were from the same class and had comparable academic backgrounds, but different demographics. All of them had knowledge of Java development (ranging from 1 to 5 years of experience), including experience in dealing with existing large software systems. In addition, Master's students had a previous experience in comprehending and maintaining existing systems, and all of them have spent an internship period in industry.

6.2.3 Research Questions

In the context of our study we address the following research questions:

- **RQ**$_1$: *How do the labels provided by automatic techniques overlap with labels produced by humans?* This is the main research question of our study, and it aims at quan-

[4]http://www.research.avayalabs.com

tifying the performance of an automatic technique when used to identify the keywords that can be used to characterize a source code artifact. In presence of a high overlap, it can be argued that the automatic technique reflects the mental model of developers when identifying keywords in source code during a comprehension task.

- **RQ**$_2$: *What code elements are often used by humans when labeling a source code artifact?* This research question is an investigation on the developer's cognitive process when reading source code. Given the artifact labels produced by humans, the research question investigates which elements of source code and comments were used to produce the labels. We determine whether words constituting labels come from class, method, or statement level comments, class names, method names, return types and parameters, attribute names, local variables, programming language keywords and native types.

- **RQ**$_3$: *What co-factors influence the effectiveness of automatic source code labeling techniques?* Our third research question aims at identifying specific characteristics of source code artifacts—such as entropy for distribution of terms—that could inhibit the performance of the automated technique in producing labels similar to those produced by humans.

6.2.4 Experimental Procedure

The study was organized in three steps, preceded by a training phase. In the first step, we asked developers to describe the selected source code classes with a set of up to 10 keywords (but not necessarily 10). Then, we applied different IR techniques and heuristics to automatically extracting keywords from the selected classes. Once the set of keywords identified by the experiment participants and the set of keywords identified by the automatic techniques were collected, we computed the overlap between them, and performed the various kinds of analyses needed to answer the research questions of our study. In the next subsections we provide details for each of these steps.

Step 0: Subjects' Training

Before asking subjects to label software artifacts, we made them familiar with the objects of the study. We provided the subjects with access to both systems a month before sending

the questionnaire. In addition, during the experiment students periodically met system experts to enrich their domain knowledge, and one of the authors (instructor of the course) also participated to the meetings to check the learning progress. Then, we presented the experimental procedure to be followed, to make each subject aware of the exact sequence of steps to perform. However, to avoid any bias, we made sure subjects were not aware of the research questions of our study.

Step 1: Human Labeling of Software Artifacts

The experiment was conducted offline, i.e., by sending the experimental material to the subjects and asking them to return the result after a given period of time. Specifically, we sent to each subject two spreadsheet files (one for each object system) containing a questionnaire to be filled-in. Each spreadsheet file consists of eleven sheets. The first one aims at collecting subject's demographics, i.e., years of computer science schooling and years of programming experience with Java. The other ten sheets aim at collecting the keywords for each of the classes to be analyzed. Each sheet reports the full class name and requires the subject to provide a list of at most ten keywords, i.e., terms considered relevant in describing the class. A term could be any source code identifier or any word contained in a compound identifier (e.g., *createFileName*) or comments. In addition, we asked subjects to provide for each class the time spent to identify the keywords, and rate the difficulty encountered to identify them on a Likert scale of 1 to 5, where 1 indicates low difficulty and 5 high difficulty.

Subjects had two weeks to fill-in the questionnaire. This point deserves further discussion because the obvious alternative would have been performing the study on-line during a limited-time laboratory. On the one hand, the latter would have given us a higher level of control over the study settings, making sure all subjects worked under the same condition, and also making sure that the information they provided us about the time spent to perform the task was correct. On the other hand, our priority was to make sure subjects produced reliable labels, i.e., they did not produce poor labels because of lack of time or inadequate code understanding. For this reason, we opted for an off-line study.

Once collected all the questionnaires, we analyzed them to identify the keywords most used to label each class. In particular, for each class C_i, we first defined the set of unique terms $T_{C_i} = \{t_1, \ldots, t_m\}$ identified by the subjects to describe C_i. For each term $t_j \in T_{C_i}$ we computed its level of agreement (LoA) as follows:

$$LoA_{C_i}(t_j) = \frac{f_{t_j}}{ns}\%$$

where f_{t_j} represents the frequency of the term t_j, i.e., the number of subjects that used t_j to

label C_i, and ns represents the number of subjects involved in our study, i.e., 17 in Experiment 1 and 21 in Experiment 2. The terms having a LoA higher than 50% (i.e., the terms selected by at least half of the subjects) represent the set of keywords (K_{C_i}) identified by subjects to label the class C_i. As it should be clearer later, the purpose of this aggregation is to produce "aggregated labels" comprising words selected by the majority of subjects, and to use such aggregated labels when understanding the provenance of words ($\mathbf{RQ}_1$), when comparing the overlap with automatic techniques ($\mathbf{RQ}_2$), and when analyzing the effect of co-factors on the effectiveness of automatic labeling techniques ($\mathbf{RQ}_3$). In addition, in $\mathbf{RQ}_2$ we also analyzed the overlap between automatically produced labels and labels produced by each subject separately.

Step 2: Automatic Labeling of Software Artifacts

In the second step of our study, we automatically identified the sets of keywords that could be used to label the selected classes. To identify such keywords, we used three different IR techniques, namely VSM, LDA, and LSI, plus three customized heuristics extracting words from specific source code elements.

VSM [80] aims at representing documents involved in an IR process as vectors in a m-dimensional space, where m is the size of the documents vocabulary. Documents can be represented as a $m \times n$ matrix (called *term-by-document matrix*), where n is the number of artifacts in the repository. A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the i^{th} term in the j^{th} document [80].

LSI [153] is an extension of the VSM. It was developed to overcome the synonymy and polysemy problems, which occur with the VSM model [153]. In LSI the dependencies between terms and between artifacts, in addition to the associations between terms and artifacts, are explicitly taken into account. For example, both "car" and "automobile" are likely to co-occur in different artifacts with related terms, such as "motor" and "wheel". To exploit information about co-occurrences of terms, LSI applies Singular Value Decomposition (SVD) [154] to project the original term-by-document matrix into a reduced space of concepts, and thus limit the noise terms may cause. Basically, given a term-by-document matrix A, it is decomposed into:

$$A = T \cdot S \cdot D^T$$

where T is the term-by-concept matrix, D the document-by-concept matrix, and S a diagonal matrix composed of the concept eigenvalues. After reducing the number of concepts to k, the matrix A is approximated with $A_k = T_k \cdot S_k \cdot D_k^T$.

Latent Dirichlet Allocation (LDA) [155] fits a generative probabilistic model from the

term occurrences in a corpus of documents. The fitted model is able to capture an additional layer of latent variables which are referred to as topics. Basically, a document can be considered as a probability distribution of topics—fitting the Dirichlet prior distribution—and each topic consists of a distribution of words that, in some sense, represent the topic.

To apply VSM, LSI, and LDA we first extracted words from source code, by removing special characters, English stop words, and (Java) programming language keywords. Each remaining word is then split using the camel case splitting heuristic. Then, we performed a *morphological analysis* to bring back words to the same root, e.g., by removing plurals from nouns, and verb conjugations. The simplest way to do morphological analysis is by using a stemmer, e.g., the Porter stemmer [156]. In the context of our study, we considered two kind of corpora: (i) words from source code including comments and (ii) words from comments only.

Then, we weighted words using two possible indexing mechanisms:

[1] *tf* (term frequency), which weights each words i in a document j as:

$$tf_{i,j} = \frac{rf_{i,j}}{\sum_{k=1}^{m} rf_{k,j}}$$

where $rf_{i,j}$ is the raw frequency (number of occurrences) of word i in document j.

[2] *tf-idf* (term frequency-inverse document frequency) which is defined as $tf\text{-}idf_{i,j} = tf_{i,j} \cdot idf_i$ where $tf_{i,j}$ is the term frequency defined above and idf_i (inverse document frequency) is defined as:

$$log \frac{n}{df_i}$$

where df_i (document frequency) is the number of documents containing the word i. *tf-idf* gives more importance to words having a high frequency in a document (high *tf*) and appearing in a small number of documents, thus having a high discriminant power (high *idf*).

For what concerns VSM, after weighting words using *tf* or *tf-idf*, for each class we selected the h words (we chose $h = 10$) having the highest weights.

As for LSI, we applied it on each single artifact (i.e., class) rather than on the corpus composed of all the classes of the system. This was done by considering the textual corpus composing the body of each method as a document in the document-by-term space, then projecting the document-by-term space into a document-by-concept space, reducing the number of concepts to k. This is because we would like to precisely identify topics representing a given class, each of them consisting in distribution of words from the class itself (and thus

not containing words belonging to other classes)[5]. For the choice of k we used the heuristic proposed by Kuhn *et al.* [147] that provided good results when labeling source code artifacts, i.e., $k = n \cdot m^{0.2}$. After that, we multiplied again the three matrices T_k, S_k and D_k^T, obtaining a term-by-document matrix A_k where term weights have been projected into the LSI space. Once the matrix A_k was computed, we extracted the h words having the highest weights in the LSI space (i.e., A_k).

LDA was applied in the same way as LSI, i.e., by building a document-by-term space over the textual corpus of methods belonging to the class to be labeled, and then applying LDA over such a space. A crucial issue in the application of LDA is choosing the number of topics. We started by setting it equal to the number of class methods (excluding getters and setters), thus assuming that each method has a specific behavior and hence brings a topic to the experiment (as done by Gethers *et al.* [149]), then we reduced it to half the number of methods, and finally we considered the extreme case of two topics only. Once LDA has been applied, we labeled each class using two heuristics:

- *core topic*: the class is labeled by the h words of the topic having the highest probability in the obtained topic distribution;
- *core words*: all the words characterizing the extracted topics are considered, and ranked according to their probability in the obtained topic distribution. The top-h words are then used to label the class. In this way we can label a class using words belonging to different topics.

In addition to the IR methods above, we also use simple heuristics based on the conjecture that when labeling a class, developers are prone to give more emphasis to terms composing the class high-level structure. Based on such a conjecture, we label a class using three different heuristics:

[1] The first one considers only terms from class name, signature of methods, and attribute names. In other words, it considers elements from the class design, whose names represent a description of the class state and behavior.

[2] The second one considers the class-level comments (excluding licensing and copyright information)—similarly to what done by Haiduc *et al.* [146, 157]. The rationale here is that the class-level comments provide a meaningful description of the class itself.

[5]When applying LSI, a generic entry (i, j) of the term-by-document matrix that is zero before the application of SVD (indicating that the term i does not occur in the document j), can assume a value different to zero after the space reduction (indicating that even if the term i does not occur in the document j it has some importance in the LSI space for the document j).

[3] The third one is a combination of the first two.

Finally, to obtain the labels, we selected the most representative words by ranking them using their *tf* and *tf-idf*, however always considering the words contained in the class name as part of the top-h words. This is because our conjecture is that the very first words a developer would use to describe a class are the words composing the class name itself.

6.2.5 Analysis Method

In the following subsection we detail on how we analyzed the experimental results to address the research questions formulated in Section 2.2.1.

Addressing $\mathbf{RQ_1}$: Computing the overlap between automatically- and human-generated labels

To address $\mathbf{RQ_1}$ we determined to what extent keywords identified by subjects correspond to those generated by automatic techniques. To this aim, we computed the overlap between them using an asymmetric Jaccard overlap [80]. Formally, let $K(C_i) = \{t_1 \dots t_m\}$ and $K_{m_i}(C_i) = \{t_1 \dots t_h\}$ be the sets of labels of class C_i identified by the subjects and the technique m_i, respectively. The overlap is computed as follows:

$$overlap_{m_i}(C_i) = \frac{|K(C_i) \cap K_{m_i}(C_i)|}{K_{m_i}(C_i)}$$

It is worth noting that the size of $K(C_i)$ might be different from the size of $K_{m_i}(C_i)$. In particular, while the number of keywords identified by an automatic technique is always 10 (by construction we set $h = 10$), the number of keywords identified by subjects could be more or less than 10 (depending on the level of agreement). For this reason, we used an asymmetric Jaccard to not penalize too much an automatic method when the size of $K(C_i)$ is higher than 10.

We statistically tested the presence of a significant difference among overlaps obtained for different labeling techniques using the Wilcoxon paired test (a paired test is necessary because we pairwise compare the overlap between the automatic and manual labeling). We used a two-tailed test to observe the differences in both directions; that, is, we do not know a priori which technique works better. Since we applied the Wilcoxon test multiple times, we had to adjust p-values. To this aim, we used the Holm's correction procedure [158]. This procedure sorts the p-values resulting from n tests in ascending order of values, multiplying the smallest

by n, the next by $n-1$, and so on. Results are interpreted as statistically significant at $\alpha = 5\%$.

In addition to the statistical comparison, we computed the effect-size of the observed differences using Cliff's delta (d) non-parametric effect size measure [82], defined as the probability that a randomly-selected member of one sample has a higher response than a randomly selected member of a second sample, minus the reverse probability. Cliff's d ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Finally, we checked whether results are consistent between the two experiments, by considering the effect of the two variables *Method* (i.e., the IR technique or the heuristic used), *Experiment* (i.e., whether results pertain to Experiment 1 or Experiment 2), and their interaction on the dependent variable *Overlap*. This was done using a two-way permutation test [159], which is a non-parametric equivalent of the two-way Analysis of Variance (ANOVA). Differently from ANOVA, the permutation test does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the test statistic while rearranging the labels (representing the various factors being considered) of the data points. We use the implementation available in the *lmPerm* R package. We set the number of iterations of the permutation test procedure to 500,000. Since this test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to choose a high number of iterations such that results did not vary over multiple executions of the procedure.

Addressing $\mathbf{RQ}_2$: Determining the origin of human-generated labels

To address $\mathbf{RQ}_2$, we manually analyzed the labels produced by the subjects and identified the code elements the words were taken from. The aim of this analysis is to investigate what elements of a source code file subjects used more in their labels, and consequently understand why different IR techniques and especially different heuristics considered in $\mathbf{RQ}_1$ exhibit different performance. Specifically, we considered the following source code elements:

- different kinds of comments, namely class level comments, method-level comments (e.g., Javadoc), and inline (statement) comments;
- different elements of method signatures, namely method names, return types, parameter names and types;
- attribute names and types;

- local variable names; and
- programming language keywords and pre-defined types, including Java pre-defined types such as *String*.

To determine the most likely used origins, for each source code file we used the oracle computed as explained in *Step 1* of our procedure and identified the element(s) each word is located in. Clearly, some words appeared in multiple locations (e.g., both in class names and comments). In such cases, the word was counted for both elements. Specifically, let $Terms(C_i, element) = \{t_1 \dots t_l\}$ be the terms extracted from the specific $element$ of class C_i. We computed the percentage of terms extracted from each source code element that were also present in the oracle ($SrcTermsInOracle$):

$$SrcTermsInOracle(C_i, element) = \frac{|Terms(C_i, element) \cap K(C_i)|}{|K(C_i)|}$$

Then, to have a better picture of the behavior of developers when labeling source code, we computed the percentage of terms in the oracle that also appeared in each source code element: ($OracleTermsInSrc$)

$$OracleTermsInSrc(C_i, element) = \frac{|Terms(C_i, element) \cap K(C_i)|}{|Terms(C_i, element)|}$$

The rationale behind of these two indicators can be explained as follows:

- $SrcTermsInOracle$ provides information about where do terms in the oracle come from,
- whereas $OracleTermsInSrc$ indicates the percentage of terms from a source that were deemed useful to produce the labels.

Addressing RQ$_3$: Factors Influencing the Accuracy of Automatic Labeling

To address **RQ**$_3$, we analyzed whether various characteristics of the source code artifacts could influence the overlap. First, we evaluated whether different labeling techniques might be affected by the specific characteristics of the analyzed documents. To this aim, we measured the entropy for distribution of terms in the class. Formally, let $t = \{t_1, \dots, t_m\}$ be the terms extracted from the class C_i. We can compute the term entropy of class C_i as follows:

$$H(C_i) = \sum_{j=1}^{m} \frac{tf_j}{n} \cdot log\left(\frac{n}{tf_j}\right)$$

where tf_j represents the frequency of the term t_j, and $n = \sum_{k=1}^{m} tf_k$. Since $H(C_i)$ ranges between 0 and $log(m)$ we normalized it as $\widetilde{H}(C_i) = H(C_i)/log(m)$.

To better interpret such entropy, it is inspired from the Shannon's definition of entropy [160], a measure of the uncertainty associated with a random variable which quantifies the information contained in a message produced by a *data emitter*. In our case, a class with a low entropy is a class where there are few dominant terms, i.e., terms with a high *tf*. Classes with a high entropy have terms with a uniformly distributed *tf*, hence labeling may become problematic. In such cases, techniques such as LSI or LDA may not effectively work to cluster artifacts. Once analyzed the entropy distribution, we divide classes in those with high entropy (above median) and low entropy (below median), and analyze the performance of the various techniques on clesses belonging to the "high" or "low" group.

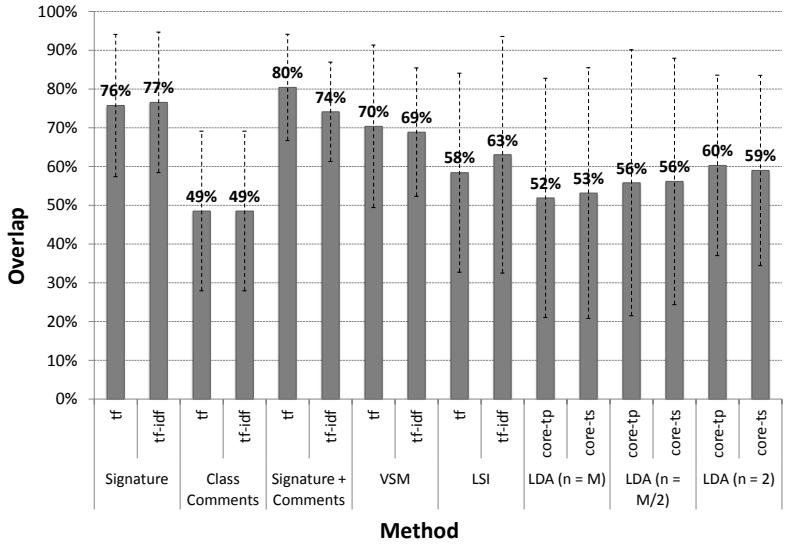
Finally, from a statistical point of view, the effect of the entropy was analyzed by means of a two-way permutation test [159]

6.3 Analysis of the Results

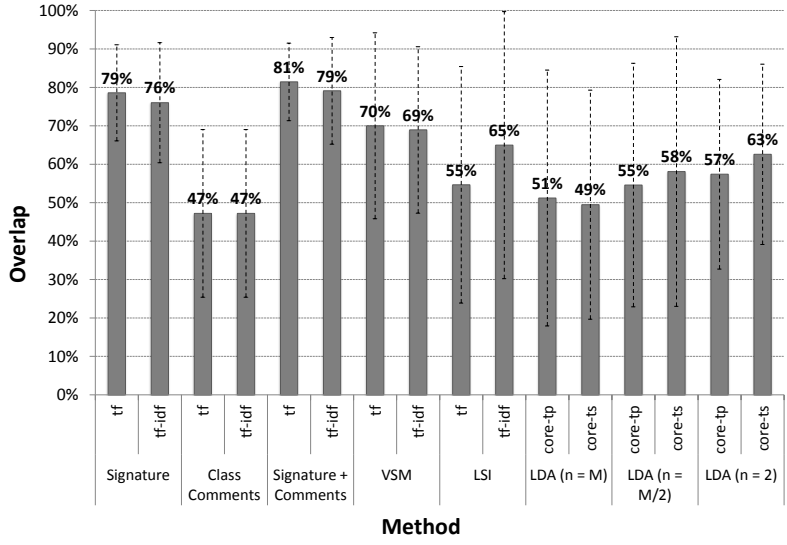
This section discusses the results of our experiments aiming at answering the research questions formulated in Section 2.2.

6.3.1 RQ_1: How do the labels provided by automatic techniques overlap with labels produced by humans?

Before comparing the labels produced by human subjects with those automatically generated by IR techniques, we analyze the agreements among the manually-produced labels. Fig. 6.3 shows—for the two systems and for the two experiments separately—the cumulative distribution of agreement among subjects. Specifically, each bar indicates that x% (value indicated on the x axis) of the subjects agree on at least y% (bar value) of the overall set words used to label the classes belonging to that system. We can note that, when considering an agreement between 40% and 60%, the percentage of the overlapped words does not change so much. The variation is higher when considering values above and below this range. This confirms that, when producing an aggregated oracle by considering the *LoA*, a threshold of 50%, as discussed in Section 6.2, represents a reasonable choice. Moreover, to inspect the consistency of the words choice by the subjects of the two experiments, Exp. I and Exp II, we also compute the overlap between the respective "oracle" summaries (6.2 and 6.3). Observing Table 6.3 for JHotDraw the words suggested by the subjects of Exp. I (Bachelor Students), they produced an "oracle" summary with a more higher number of words if compared with

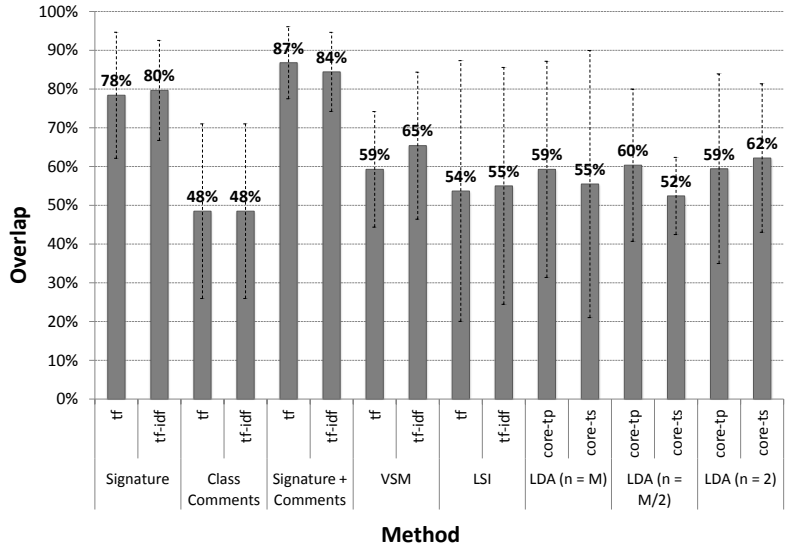


(a) Experiment 1

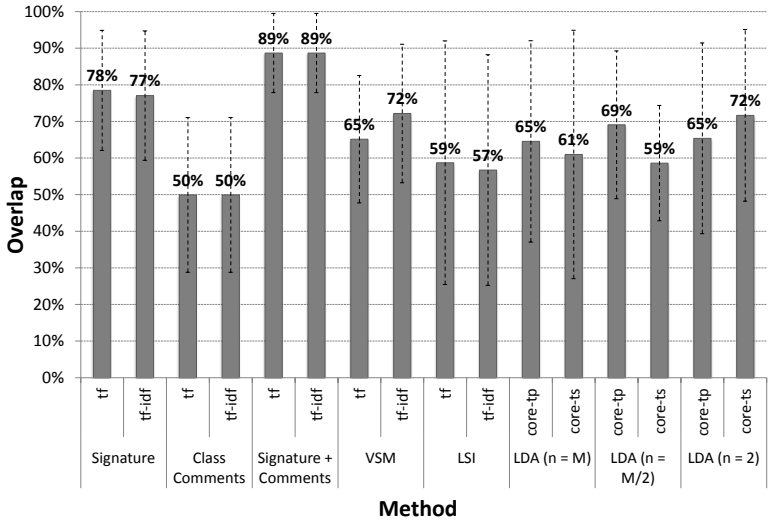


(b) Experiment 2

Figure 6.1: eXVantage: Mean overlap between automatically-produced labels and manually-generated labels.

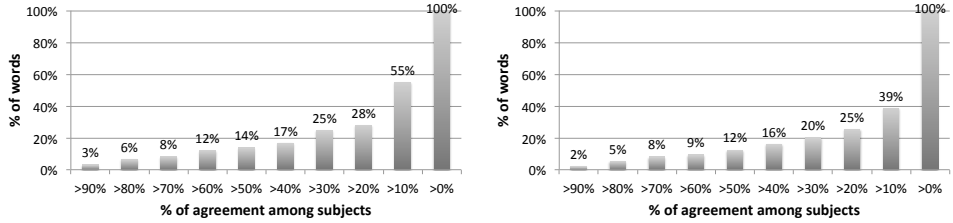


(a) Experiment 1



(b) Experiment 2

Figure 6.2: JHotDraw: Mean overlap between automatically-produced labels and manually-generated labels.



(a) JHotDraw: agreement variation among subjects for Experiment 1

(b) JHotDraw: agreement variation among subjects for Experiment 2

Figure 6.3: Cumulative distribution of agreement among subjects.

the number of words suggested by the subjects of Exp. II (Master Students). Instead, for eXVantage the size of the two "oracle" summaries is roughly the same. Moreover, the overlap between the "oracle" summaries used for the two experiments is very high and at least the 76% for both the two systems considered in the study and in both the experiments confirming that there is a big core set of *common words* between subjects of Exp. I and Exp II (i.e., Bachelor Students and Master Students).

Fig. 6.1 and Fig. 6.2 show—for eXVantage and JHotDraw respectively—the average overlap between the human-generated labels and the automatic labels obtained (i) using different IR methods (VSM, LSI, and LDA), (ii) the heuristic that extracts words from class interfaces (referred as "signature"), (iii) the heuristic that extracts words from class-level comments (referred as "class comments") (iv) the heuristic that combines the previous two heuristics (referred as "signature + comments"). The figures also show the variability (standard deviation) of the overlap obtained by the different methods over different classes. The analysis reveals that the overlap achieved by all the experimented methods for automatic labeling varies between 40% and 90%. In particular, results indicate that—in both experiments and for both systems—LSI and LDA achieve the worst overlap if compared to the simpler VSM (which, unlike LSI and LDA, does not reduce the term space into a topic/concept space), and to the signature and signature+comments heuristics. Specifically, the different variants of LSI and LDA achieve an average overlap with human labeling varying between 40% and 65%, whereas VSM obtains an average overlap about 3% higher than what achieved by LSI and LDA. The best performance are achieved by the simple heuristics (signature and signature+comments): in such cases the overlap is always higher than 75% for eXVantage and 85% for JHotDraw. Instead, considering comments only does not guarantee good performance (the overlap is always smaller than 50%). In essence, a heuristic that considers class comments to automatically produce labels is not sufficient to achieve good performance. Our interpretation of this result is the following: taking the most frequent words from class-level

Table 6.2: eXVantage: Overlap between the "oracle" summaries used for the two experiments

eXVantage		
Experiment 1	Experiment 2	Overlap
63	62	49

Table 6.3: JHotDraw: Overlap between the "oracle" summaries used for the two experiments

eXVantage		
Experiment 1	Experiment 2	Overlap
75	60	57

comments would likely pick random words, because it is difficult to build a vector space model on such a small corpus. For this reason, automatically generating a label from class comments does not produce satisfactory results.

In summary, seems that simple heuristics (signature and signature+comments) experimented in this study are the more precise in generate *high quality summaries* of source code to help newcomers in program comprehension os source code elements. With the aim of statistically supporting this finding , Table 6.4 reports the Cliff's *d* values obtained in pairwise comparison of the various labeling methods, highlighted in bold face when p-values (adjusted using the Holm's correction) of the Wilcoxon Rank Sum test are statistically significant (i.e., when the adjusted p-value < 0.05). Since it would not be practical to show results related to all possible comparisons, we only compared the top-two performing heuristics (i.e., signature and signature+comments) with all other techniques. Results of the statistical tests and effect size measures confirm our preliminary findings discussed above. The signature+comments heuristic produces a statistically significant improvement with respect to other automatic methods in the majority of cases (38 cases out of 40). Moreover, the Cliff's *d* effect size is large in 73% of the cases (29 out of 40), medium in 20% of the cases (9 out of 40), and small in 5% of the cases (2 out of 40). However, it is important to note that for the few cases (2 cases out of 80) for which there is no statistical difference—e.g., when comparing *Signature* with *VSM-tf-idf* on eXVantage and JHotDraw—the adjusted p-values range between 0.05 and 0.06, thus they are marginally significant. Moreover, the correspondent Cliff's d is positive (medium in some cases, small in other cases), highlighting an improvement for these cases too. The heuristic based on the signatures (without taking into account the class comments) also achieves a significant improvement with respect to other heuristics in the majority of cases (38 cases out of 40). The Cliff's *d* effect size value is large in 57% of the cases (23 out of 40), medium in 30% of the cases (12 out of 40), and small in 5% of the

Table 6.4: Cliff's d for differences of overlap between automatic labeling and human labeling provided by each subject. Values shown with $*$ for comparisons where the Wilcoxon Rank Sum test indicates a significant difference. We use S, M, and L to indicate a small, medium and large effect size, respectively.

Comparison	Experiment 2		Experiment 1	
	eXVantage	**JHotDraw**	**eXVantage**	**JHotDraw**
Sign. + class com. > VSM-tf	0.62* (L)	0.31* (S)	0.50 *(L)	0.45* (M)
Sign. + class com. > VSM-tf-idf	0.48* (L)	0.29$^+$ (S)	0.34$^+$ (M)	0.39 (M)
Sign. + class com. > LDA-core-tp (n=M)	0.97* (L)	0.39* (M)	0.89* (L)	0.51 * (L)
Sign. + class com. > LDA-core-ts (n=M)	0.94 *(L)	0.61 *(L)	0.90 *(L)	0.73 * (L)
Sign. + class com. > LDA-core-tp (n=M/2)	0.80 *(L)	0.41 *(M)	0.79* (L)	0.45* (M)
Sign. + class com. > LDA-core-ts (n=M/2)	0.89* (L)	0.56 *(L)	0.82* (L)	0.68* (L)
Sign. + class com. > LDA-core-tp (n=2)	0.65 *(L)	0.45* (M)	0.58* (L)	0.41* (M)
Sign. + class com. > LDA-core-ts (n=2)	0.66 *(L)	0.43* (M)	0.53 *(L)	0.54* (L)
Sign. + class com. > LSI-tf	0.80 *(L)	0.72* (L)	0.61* (L)	0.76 *(L)
Sign. + class com. > LSI-tf-idf	0.67* (L)	0.69 *(L)	0.54* (L)	0.79* (L)
Sign. > VSM-tf	0.50 *(L)	0.26 *(S)	0.39 *(M)	0.38* (M)
Sign. > VSM-tf-idf	0.36 * (M)	0.25 * (S)	0.23 $^+$ (S)	0.33 $^+$(M)
Sign. > LDA-core-tp (n=M)	0.91* (L)	0.32 *(S)	0.86 * (L)	0.41 * (M)
Sign. > LDA-core-ts (n=M)	0.88 * (L)	0.54 *(L)	0.84* (L)	0.66 *(L)
Sign. > LDA-core-tp (n=M/2)	0.73* (L)	0.27 *(S)	0.70* (L)	0.36* (M)
Sign.> LDA-core-ts (n=M/2)	0.80* (L)	0.44* (M)	0.78* (L)	0.60* (L)
Sign. > LDA-core-tp (n=2)	0.51* (L)	0.37 *(M)	0.48 *(L)	0.34 *(M)
Sign. > LDA-core-ts (n=2)	0.51 *(L)	0.35* (M)	0.45* (M)	0.46* (M)
Sign. > LSI-tf	0.71* (L)	0.61 *(L)	0.55* (L)	0.69 *(L)
Sign. > LSI-tf-idf	0.55 *(L)	0.60 *(L)	0.48 *(L)	0.71 *(L)
Sign. + class comm. > Sign.	0.22* (S)	0.06	0.27 *(S)	0.08

cases (4 out of 40). In summary, the signature+comments heuristic outperforms the heuristics considering class signature and comments separately. A direct comparison between the two heuristics shows that there is a significant difference between them only for eXVantage where, as discussed above, comments seem to be able to improve the performance of the signature-based heuristic.

As also shown in Fig. 6.1 and Fig. 6.2, it is worthwhile to point out how, $\mathbf{RQ}_1$ results are pretty consistent between Experiment 1 and Experiment 2. This is also confirmed by a two-way permutation test (where we consider the overlap as dependent variable, and both the IR method used and the experiment as independent variables), which for both JHotDraw and eXVantage indicated that the overlap is significantly influenced by the Method (p-value < 0.001), while it is not influenced by the Experiment (p-value$= 1$) nor by the interaction of the Method and the Experiment (p-value$= 1$).

Table 6.5: $SrcTermsInOracle$: Percentage of oracle words belonging to different source code entities, and (in parenthesis) $OracleTermsInSrc$: percentage of entity words considered in the oracle.

System	Exp.	Class comm.	Method comm.	Inline comm.	Class name	Attrib.	Param.	Method name	Returned type	Local var.	Keywords and predef. types
eXVantage	Exp 1	44%	46%	17%	48%	29%	52%	43%	21%	37%	0%
		(26%)	(8%)	(12%)	(79%)	(24%)	(38%)	(33%)	(59%)	(39%)	(0%)
	Exp 2	35%	66%	15%	26%	34%	29%	48%	0%	15%	5%
		(21%)	(12%)	(10%)	(80%)	(28%)	(20%)	(37%)	(0%)	(15%)	(3%)
JHotDraw	Exp 1	51%	58%	32%	25%	34%	46%	62%	4%	30%	0%
		(15%)	(14%)	(20%)	(76%)	(60%)	(30%)	(39%)	(9%)	(23%)	(0%)
	Exp 2	60%	68%	35%	30%	35%	50%	50%	3%	32%	0%
		(14%)	(13%)	(17%)	(72%)	(49%)	(25%)	(25%)	(6%)	(19%)	(0%)

RQ_1 Summary: The achieved results indicate that the best heuristics for class labeling are simple heuristics considering class signatures and combining class signatures with class comments, while class comments alone do not perform well.

6.3.2 RQ_2: What code elements are often used by humans when labeling a source code artifact?

Table 6.5 reports $SrcTermsInOracle$, the percentage of terms in the human-generated oracle (obtained by computing the level of agreement, as explained in Section 6.2) appearing in various source code entities. Also, the table reports (in parentheses) $OracleTermsInSrc$, the percentage of terms belonging to that source code entity that were actually used in the oracle.

The obtained results highlight that the majority of the terms suggested by the subjects belong to *method names* (ranging between 43% and 62%), *method parameters* (ranging between 29% and 52%), *class comments* (ranging between 35% and 60%) and *method comments* (ranging between 46% and 68%). The percentage of labels taken from terms composing a class names is relatively lower (between 25% and 48%) than what obtained for method names and parameters, however, this only happens because class names are composed of few words, that do not suffice to properly describe the responsibility of the class itself. However, as the percentages in parentheses show, over 70% of terms belonging to class names are used in the oracles, confirming that terms from class names are almost always taken when producing labels. We can also notice that, for eXVantage, the percentage of labels taken from terms composing class names is higher in the first experiment than in the second one. There is no straight-forward explanation for that. However, it might have happened that Experiment 1 subjects (undergraduate students) usually kept terms from class names as first and obvious

choice, whereas Experiment 2 subjects (graduate students) also looked elsewhere, and this happened in particular for eXVantage than for JHotDraw, as the former has more comments than the latter.

Thus, terms describing the class interface (i.e., coming from method names and parameters) result to be particularly useful for the labeling process. The same can be said for class-level comments and method comments: this is quite intuitive because class-level comments often contain short descriptions of the class and method behavior and responsibilities. This finding is fully in agreement with results of the study done by Haiduc *et al.* [146], who used the first words of a source code file—often corresponding to class-level comments—to summarize a class.

When looking at $OracleTermsInSrc$ for class and method comments, we can notice they are relatively low (below 30%). This indicates that, while oracles contain a high percentage of class and method comments, such comments also contain many words that do not appear in the oracle, which can be expected considering the verbosity of comments if compared to a 10-words label. This has a clear, negative, consequence if trying to use comments alone to produce labels (see results of $\mathbf{RQ}_1$ in Fig. 6.1 and Fig. 6.2). In essence, although words contained in class and method comments are useful to produce the label, the automatic heuristic is not able to discern the relevant words from the noise.

We found that terms taken from method local variables are relatively less used (between 15% and 37%), likely because they represent low-level entities that are, generally, not particularly relevant when describing the overall responsibilities of a class. Similar considerations apply for inline comments. Finally, method return types and programming language predefined types and keywords are almost never used. Again, such entities could be used when describing methods (where one could mention algorithmic details or returned values) while they are not needed when describing a class.

> **$\mathbf{RQ}_2$ Summary:** We can conclude that, to label a class, subjects participating to our experiments mainly used class and method names and signatures, as well as some words of class and method comments. However, comments also contain a high percentage of words not appearing in the oracle. This confirms why the heuristics signature and signature+comments produces very good results, while comments alone does not.

6.3.3 $\mathbf{RQ}_3$: What co-factors influence the effectiveness of automatic source code labeling techniques?

In the following, we analyze how the characteristics of the source code artifacts influence the performance of automatic labeling approaches. First, we study how such performance

are related to the term entropy in the artifacts. Then, we investigate how the performance correlate with the class size and comment verbosity, and how such factors correlate to the difficulty for labeling such classes, measured in terms of the effort our subjects spent in the labeling tasks.

Effect of term entropy

Results of $\mathbf{RQ}_1$ The achieved results indicate that the best heuristics for class labeling are simple heuristics considering class signatures and combining class signatures with class comments. LSI and LDA are based on clustering analysis: all documents (classes in our case) are implicitly clustered on the basis of their shared latent concepts/topics. Source code artifacts having the same latent concepts/topics are considered as components of the same cluster, while artifacts having different latent concepts/topics are placed in different clusters. For this reason, such techniques are often used to identify topics in source code (see for example the work of Kuhn *et al.* [147]). Thus, LSI and LDA work well if documents contain dominant terms—i.e., terms having a higher frequency than others—that can be used to derive and extract the topics discussed in the documents. More formally, LDA and LSI work well for documents having a low term entropy, i.e., documents for which the distribution of terms—proportional to their frequency—is not uniform across different documents, with few terms that are more frequent than others. In essence, because of the non-uniform distribution of terms—and their low entropy—it is easier for the clustering-based techniques to derive the dominant concepts/topics discussed in the documents.

Differently from traditional natural language artifacts, heterogeneity is not always present in source code artifacts, especially when the goal is to extract topics from single classes, composed of "conceptually cohesive" [161] methods. A possible explanation is that a class is a crisp abstraction of a domain/solution object, and should have a few, clear, responsibilities. Thus, a class has generally a few number of strongly-coupled topics. Moreover, the frequency of terms contained in the source code is very low. This means that when analyzing the textual content of classes, it is difficult to identify dominant terms that characterize the topics discussed in the class itself. To verify such a conjecture, we computed the entropy of terms contained in the classes used in our study as described in Section 6.2. High entropy indicates that the probability distribution of the terms is quite uniform, reducing the capability of clustering-based techniques to identify dominant terms that can be used to characterize the class topics.

Fig. 6.4 shows boxplots of the term entropy for the classes considered in our study. For both systems, the median term entropy (and even the first quartile) is greater than 0.8, indicating that the selected classes have a quite high entropy. Since a high entropy means that terms

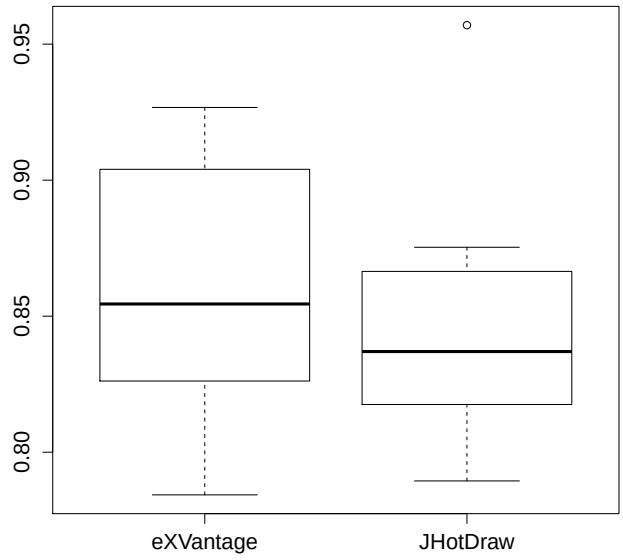
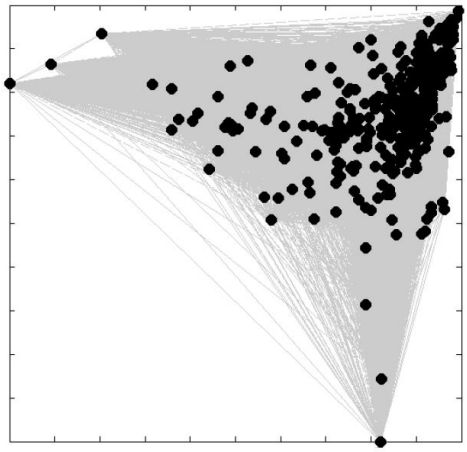


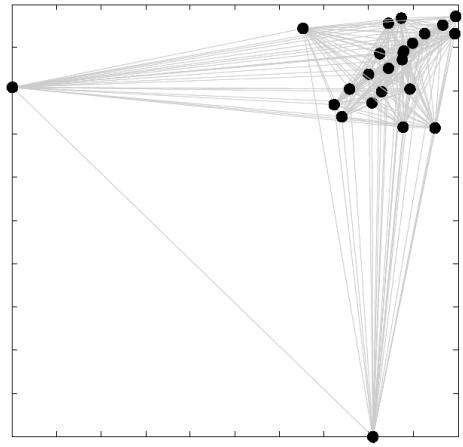
Figure 6.4: Entropy of terms in the classes sampled for our experiments.

occurring in a class have almost the same probability, then it is hard to identify dominating terms that can be used to label such a class. This explains the poor performance of the two clustering-based IR methods, namely LSI and LDA.

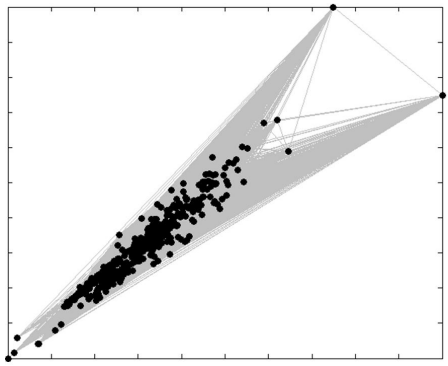
It is important to note that the classes selected in our study are not the only ones having a high entropy. We also computed the entropy for all classes of the two systems. We obtained an average entropy of 0.89 for ExVantage, and 0.87 for JHotDraw. Thus, we can conclude that clustering-based approaches will lead to some difficulties when dealing with highly-homogeneous collections, such as source code classes. To provide further evidence to this issue, Fig. 6.5 displays pairwise document distances achieved using LDA when clustering source code artifacts of eXVantage at two different levels of granularity. For eXVantage, Fig. 6.5-(a) shows the results achieved by clustering all eXVantage classes, while Fig. 6.5-(b) shows the clustering of methods of a class, EventThread (such a class was just taken as an example to show the distribution of topics among methods). Similarly, Fig. 6.5-(c) shows the clustering of JHotDraw classes, while Fig. 6.5-(d) shows the clustering of methods belonging to the JHotDraw GraphicalCompositeFigure class. Each node of the graph represents



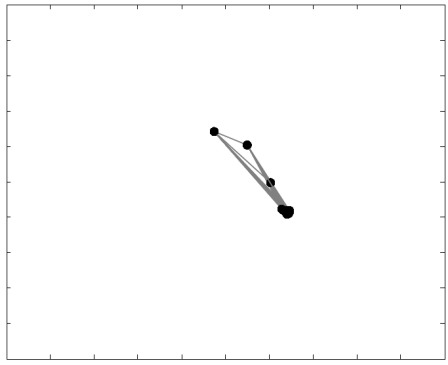
(a) eXVantage: Distance between classes



(b) eXVantage: Distance between methods of class EventThread



(c) JHotDraw: Distance between classes



(d) JHotDraw: Distance between methods of class GraphicalCompositeFigure

Figure 6.5: Distance between source code elements.

a class (or a method), while the weight of the edges measures the distance between topic distributions of each pair of classes (methods). Thus, if two classes (or methods) have different distribution of topics (i.e., they belong to different topics), then the correspondent nodes are far from each other in the graph. As we can notice from the figures, in all cases most classes (or methods) are concentrated in a small area. In such a scenario, it is quite difficult to discriminate between classes (or methods), and consequently efficiently clustering them.

To provide further evidence of the relationship between entropy and the overlap achieved

Table 6.6: Examples of high and low labeling overlap achieved on eXVantage. Terms in bold face represent the gold words used by subjects.

Class	Method	# Topics	Topic Probability	Terms	Overlap	Entropy
CFGManager	LDA core-tp	2	0.49	string, **cfg**, key, mapping, filename, temp, file, global, instrument, integer	0.20	0.97
			0.51	list, temp, string, value, **cfg**, table, **classname**, name, map, key		
EdgeElement	LDA core-tp	2	0.26	**edge**, set, **node**, **element**, name, boundary, undefined, defined, illegal, access	1	0.78
			0.74	**node**, attribute, **graph**, **edge**, boundary, **element**, doc, inherit, parameter, value		

by LDA, Table 6.6 shows two examples of eXVantage classes having high and low term entropy, respectively. The terms identified by LDA that overlap with those identified by humans are shown in bold face. For the class CFGManager, the entropy is very high (0.97), highlighting that all the terms extracted from the source code have the same frequency. In such a scenario it turns out to be difficult for LDA to identify the most important topic. Indeed, the class CFGManager has the same probability to discuss about one of the two main topics indifferently, since the two topics are quite equiprobable. This results in a very low overlap between automatic and human label. In the latter case, there are some dominating terms (as indicated by the much lower entropy) and LDA is able to identify well-distinct topics in the class under analysis. Indeed, when applying LDA, we can observe that in EdgeElement one of the two topics has a probability greater than the other one. Such terms are selected by LDA to label the class and, as suggested by the high overlap, the same terms were also selected by subjects. The factor *entropy* affects not only LDA but also LSI that is a clustering technique too. In particular, for the two classes considered in the previous example when using *tf* as terms weighting schema, LSI obtains a higher overlap with human labeling (100%) for EdgeElement, which exhibits a low term entropy, while for CFGManager the overlap is about 40%.

Similar considerations apply for JHotDraw. To this aim, Table 6.7 shows two examples of classes from JHotDraw with high and low term entropy respectively. The entropy is high for PreferenceUtil, while it is low for GraphicalCompositeFigure. Such different entropy values affect the capability of LDA to consistently identify and extract the main topic. Indeed, for PreferenceUtil LDA identifies two topics having exactly the same probability, resulting in a very low overlap with automatic and human label. Conversely, for GraphicalCompositeFigure LDA extracts two topics with two quite different probabilities. This is mirrored by

Table 6.7: Examples of high and low labeling overlap achieved on JHotDraw. Terms in bold face represent the gold words used by subjects.

Class	Method	# Topics	Topic Prob.	Terms	Overlap	Entropy
GraphicalCompositeFigure	LDA core-tp	2	0.65	**figure**, key, **presentation**, **attribute**, **composite**, value, **graphical**, set, **draw**, element	0.86	0.82
			0.35	**presentation**, **figure**, **attribute**, key, bounds, **graphical**, set, listener, event, undoable		
PreferenceUtil	LDA core-tp	2	0.50	screen, insert, name, bound, **window**, component, preferred, height, **size**, **preference**	0.38	0.87
			0.50	time, bound, event, component, **window**, screen, name, **preference**, **size**, rectangle		

a high overlap. Similar results are achieved using LSI. Indeed, using *tf* as terms weighting schema, LSI obtains better results for the class having the lowest term entropy. Indeed, the achieved overlap with human labeling is about 38% for PreferenceUtil and the 100% for GraphicalCompositeFuigure. In summary, these examples allow us to highlight the strong relationship between the entropy of terms and the performance of the clustering based automatic techniques, i.e., LDA and LSI. The higher the term entropy, the lower the performance achieved by the clustering techniques.

In order to generalize this kind of analysis for all the techniques, we investigate whether the terms entropy influence the achieved overlap, or interacts with the adopted technique to this regard. For a first analysis, we grouped the classes in two groups (*low* and *high*), based on their terms entropy. In particular, a first group contains all classes for which the terms entropy value is higher than the median, and a second cluster contains the remaining ones. Then, we analyzed the overlap similarly to what was done in $\mathbf{RQ}_2$ for low and high entropy separately. Results are shown in Table 6.8. Such results indicate all the techniques are effected by the entropy factor. For example, it can be noticed that VSM performs, on average, 10% better on classes with lower entropy. Instead, the performance of LDA are not necessarily better for classes with lower terms entropy than the other: in the 50% of cases LDA achieves a higher overlap with human labeling for classes with lower entropy values, while the scenario is opposite in the remaining cases. The proposed heuristics is also affected by the entropy factors, however, they always outperforms all the other techniques independently of the entropy of terms.

Table 6.9 reports the results od the permutation test with the aims at providing statistical support to the results shown in Table 6.8. The entropy factor plays a significant role for both systems while it shows a significant interaction only for the Experiment 1 and only on

Table 6.8: Average overlap between manual labeling and automated labeling collected by the terms entropy (*high* and *low* terms entropy). M represents the number of methods in the class.

Corpus type		exVantage				JHotDraw			
		Experiment 1		Experiment 2		Experiment 1		Experiment 2	
		Low	High	Low	High	Low	High	Low	High
Signature	**tf**	**89%**	63%	81%	76%	75%	82%	74%	83%
	tf-idf	86%	67%	74%	78%	78%	82%	74%	80%
Class Comments	**tf**	52%	45%	47%	47%	50%	47%	53%	46%
	tf-idf	52%	45%	47%	47%	50%	47%	53%	46%
Sign. + Class Comments	**tf**	**89%**	72%	**84%**	79%	**84%**	**90%**	**89%**	**89%**
	tf-idf	73%	**75%**	77%	**81%**	81%	88%	**89%**	**89%**
VSM	**tf**	82%	58%	81%	59%	59%	59%	70%	61%
	tf-idf	77%	61%	76%	61%	64%	67%	76%	69%
LSI	**core-tp**	71%	46%	71%	39%	57%	50%	63%	54%
	core-ts	80%	46%	81%	49%	41%	69%	3600%	77%
LDA(n = M)	**core-tp**	56%	48%	52%	50%	57%	62%	66%	63%
	core-ts	48%	58%	47%	52%	55%	56%	81%	41%
LDA(n = M/2)	**core-tp**	65%	46%	64%	45%	59%	62%	75%	63%
	core-ts	61%	51%	61%	55%	51%	54%	62%	55%
LDA(n = 2)	**core-tp**	72%	58%	72%	42%	54%	65%	71%	60%
	core-ts	71%	47%	70%	55%	59%	65%	80%	64%

Table 6.9: Permutation test by Method and terms entropy.

System	Exp.	Method p-value	Entropy p-value	Method:Entropy p-value
eXVantage	Exp 1	**0.04**	$<$ **0.001**	**0.02**
	Exp 2	$<$ **0.001**	$<$ **0.001**	0.14
JHotDraw	Exp 1	$<$ **0.001**	0.87	0.98
	Exp 2	$<$ **0.001**	**0.004**	0.81

eXVantage. These results confirm that the automatic labeling is more difficult for classes with high terms entropy independently of the specific automatic technique use

> **RQ_3 Summary:** Simple heuristics exhibit good performance independently of the class size and comment verbosity. Furthermore, LDA and LSI work well on artifacts having a low term entropy, which is quite infrequent for source code artifacts.

6.4 Threats to Validity

This section describes threats that can affect the validity our study. Threats to *construct validity* concern relationship between the theory and the observation. In our study, threats to

construct validity are mainly related to the measurements we performed to address our research questions. To investigate to what extent labelings identified by humans match those identified by automatic approaches ($\mathbf{RQ}_1$), we rely on a widely used IR technique, i.e., the Jaccard overlap score. For what concerns the origin of terms used by humans to label software artifacts ($\mathbf{RQ}_2$), we performed a manual analysis to map the list of terms provided by subjects onto source code elements. To avoid mistakes, the analysis was performed by two independent persons (two of the authors), and results were compared and discussed to solve inconsistencies.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Wherever appropriate, we use statistical procedures to corroborate our results. Specifically, we use non-parametric tests (Wilcoxon) and correlation procedures (Spearman rank), adjusting p-values using the Holm procedure when performing multiple tests on the same data. In addition, we report Cliff d effect size to provide a quantitative assessment of the differences found. Also, whenever a co-factor analysis is needed, we use permutation tests that, differently from ANOVA, does not require data to be normally distributed.

Threats to *internal validity* are related to factors that can influence our results. In terms of human factors, thre was no abandonment, i.e., all subjects completed the task. To mitigate the effect of labeling variability across subjects, we chose the most frequent words they used to label each class. Another factor that can influence our results could be the choice of the number of topics for LDA, and the number of concepts in LSI. For the former, we used different settings (i.e., number of topics equal to the number of methods, half of them, and two topics only). For the latter, we used the heuristics adopted by Kuhn *et al.* [147]. Furthermore, we investigate whether other factors—such as the class entropy and the textual similarity among classes—could have influenced the results of the various techniques.

Threats to *external validity* concern the generalization of results. We tried to investigate as many IR-based techniques as possible to perform automatic labeling: VSM, LDA, LSI, and three simple heuristics, considering words form class comments, signatures, and their combinations. Also, we compared different weighting schemes (*tf* and *tf-idf*), and different sources of information (source code including comments, and comments only). However, we are aware that there could be other heuristics we did not consider. We are also aware that the study involved objects from two Java systems only. Therefore, results could be different if replicating the study on other systems, and in particular on objects developed with different programming languages. For instance it would be interesting to replicate on programs written using programming languages like C where, differently from Java, term separators (e.g., Camel Case) are not consistently used, and also there is a high usage of abbreviations [162]. Last, but not least, we are aware that our labelings were performed by

students. Although we carefully avoided (by means of a proper training) that the limited knowledge of the objects could have influenced the results, we are aware that results could possibly vary if replicating with professionals. For example, it may (or may not) happen that professionals would, in some case, pick other terms than those in the method signatures/class names, thus decreasing the performance of the simple heuristic. As a partial mitigation to this threat, subjects employed in Experiment 2 had more experience than those of Experiment 1—also including industrial experience gained during periods of internship.

6.5 Related Work

The rapid development of software engineering methods and tools and the increasing complexity of software projects has led, in the last decades, to a significant production of textual information contained in structured and unstructured project artifacts. As consequence, several researchers investigated the analysis of textual information contained in the artifacts of software repositories to support activities such as impact analysis [163], clone detection [164], feature location (e.g., [165]), definition of new cohesion and coupling metrics (e.g., [161, 166]), assessment of software quality (e.g., [151, 152, 167, 168]), and traceability recovery (e.g., [169–175]).

Related to our work is the use of textual analysis, and in particular topic modeling techniques, to mine and understand topics within source code. Such approaches aim at providing support to program comprehension by deriving a snapshot of the system that is easier to understand. Clustering semantically related classes or labeling can be considered as a way to summarize the responsibilities of source code artifacts aiming at aiding developers in comprehension tasks.

Maletic and Marcus [176] proposed the combined use of semantic and structural information of programs to support comprehension tasks. Semantic information, captured by LSI, refers to the domain specific issues (both problem and development domains) of a software system, while structural information refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents. Components within a software system are then clustered together using the combined similarity measure.

Kuhn *et al.* [177] extended the work by Maletic and Marcus introducing the concept of semantic clustering, a technique based on LSI to group source code documents that share a similar vocabulary. After applying LSI to the source code, the documents are clustered based on their similarity into semantic clusters, resulting in clusters of documents that implement similar features. The authors also used LSI to label the identified clusters. Finally, a visual notation is provided, which is aimed at giving an overview of all the clusters and their

semantic relationships.

Baldi *et al.* [178] applied LDA to source code to automatically identify concerns. In particular, they used LDA to identify topics in the source code. Then, they used the entropies of the underlying topic-over-files and files-over-topics distributions to measure software scattering and tangling. Candidate concerns are latent topics with high scattering entropy.

Linstead *et al.* [179] used LDA to identify functional components of source code and study their evolution over multiple project versions. The results of a reported case study highlight the effectiveness of probabilistic topic models in automatically summarizing the temporal dynamics of software concerns.

Thomas *et al.* [180] also applied LDA to the history of the source code of a project to recover its topic evolutions. The authors considered additional topic metrics (i.e., scatter and focus) to better understand topic change events, and providing a detailed, manual analysis of the topic change events to validate the results of the approach.

All these works on topic analysis produced project-specific topics that needed to be manually labeled. Hindle *et al.* [181] presented a cross-project data mining technique leveraging software engineering standards to produce a method of partially-automated (supervised) and fully-automated (semi-unsupervised) topic labeling. Since the proposed approach is not project-specific, it is possible to use it to compare two distinct projects.

Hindle *et al.* [150] used LDA in an industrial context to relate requirements to code. They performed an empirical study in order to verify whether the information extracted with LDA matches the perception that program managers and developers have about the effort put into addressing certain topics. The results indicated that in general the identified topics made sense to practitioners and matched their perception of what occurred even if in some particular cases practitioners had difficulty interpreting and labeling the extracted topics.

Recently, Medini *et al.* [182] used IR methods and formal concept analysis to produce sets of words helping the developer to understand the concept implemented in execution traces. The authors performed both a qualitative as well as a quantitative analysis of the proposed approach. The analysis revealed that the approach is quite accurate in identifying topics in execution traces and in most cases the suggested labeling terms are effective to help grasping the segment functionality.

Besides topic analysis, summarization techniques have also been applied to source code artifacts for different purposes. Rastakar *et al.* used a Machine Learning (ML) approach to automatic generate summaries of bug reports [183] and software concerns [184]. Buse and Weimer proposed an approach to automatically generate human-readable documentation for arbitrary code differences [185].

Murphy [186] presented the software reflection model and the lexical source model ex-

traction. Such models can be considered as a lightweight summarization approach of software. Sridhara *et al.* used natural language processing techniques to automatically generate leading method comments [187], and comments for high-level actions [188]. Also the automatic generation of comments can be considered as a kind of summarization of source code components.

In summary, several of the works described above used different techniques to label or summarize software artifacts. Our study constitutes a complementary contribution to such approaches, because it aims at assessing automatic labeling techniques by comparing them with human-generated labels.

Haiduc *et al.* [146, 157] recently applied several summarization techniques for the automatic summarization of source code artifacts with the purpose of aiding developers in comprehension tasks. In a reported case study [146] they found that a combination between techniques making use of the position of terms in software and Textual Retrieval (TR) techniques capture the meaning of methods and classes better than any other of the studied approaches. In addition, an experiment conducted with four developers revealed that the summaries produced using this combination make sense.

To the best of our knowledge, the work by Haiduc *et al.* [146] is the most relevant to our work. However, while Haiduc *et al.* asked developers to validate the summaries, we compare the labelings obtained with automatic techniques with humans' labels. This provides an objective and more precise evaluation of the accuracy of automated techniques in approximating the cognitive model of developers. In addition, we used a larger and different set of techniques based on advanced IR methods and also on "ad hoc" heuristics. Finally, we involved in our experimentation a larger number of subjects (37 *vs.* 4) having different experience (both undergraduate and graduate students), showing that results are almost perfectly consistent between the two experiments. In terms of results, while we share with Haiduc *et al.* [146] findings about the importance of class-level comments for artifact labeling, our results also highlight that: (i) comments alone do not produce good labels, because they contain several words that were discarded by humans. Instead, comments are useful when combined with class signatures; and (ii) simpler heuristics considering class signatures (possibly combined with comments) outperform IR techniques.

6.6 Summary

In recent years, researchers have applied various IR methods to "label" software artifacts by means of some representative words, with the aim of facilitating their comprehension or just to better visualize them. This section reported an empirical study aimed at investigating to

what extent a source code labeling based on IR techniques would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task.

We conducted two experiments, in which we asked 17 Bachelor's Students and 21 Master's Students, respectively, to describe 10 classes taken from a software system using at most ten words extracted from the class source code and comments. Then, we analyzed (i) what kind of source code (and comment) elements were used by subjects to produce the labels; (ii) to what extent the keywords identified using various IR techniques overlap with those identified by humans; and (iii) what characteristics of the analyzed artifacts could influence the effectiveness of the various techniques used to automatically produce labels. As possible techniques, we considered VSM, LSI, LDA, and some *ad hoc* heuristics picking terms from specific parts of the source code and comments.

Results show that overall there is a relatively high overlap between automatic and human-generated labels, ranging between 50% and 90%. However, the highest overlap is obtained by using the simplest heuristic, while the most sophisticated techniques, i.e., LSI and LDA, provide generally the worst accuracy. One reason of the result is that developers mainly used words from class names, method names and signatures, and (partially) from class and method comments to label artifacts. We also found that the high entropy of terms in the classes inhibits the capability of topic modeling techniques—i.e., LSI and LDA—to efficiently identify and cluster topics in source code. This result highlights that approaches such as LDA and LSI are worthwhile of being used when analyzing heterogeneous collections, where documents can contain information about multiple topics [141]. Unfortunately, such an heterogeneity is not always present in source code artifacts. Thus, the *ad-hoc* heuristics experimented in this study represent a valid approach to *build high quality summaries* of source code elements to help new developers in program comprehension.

Part III

Recommenders

In this last part of the thesis, we discuss the possibilities to help projects newcomers in specific moment of their life in OSS projects. Specifically, when a newcomer joins a software project she needs to be trained from several points of view, (i) technical skills, (ii) projects organization, or (iii) implementation details.

In Chapter 7 we propose an approach, named YODA (**Y**oung and newc**O**mer **D**eveloper **A**ssistant) aimed at identifying and recommending mentors in software projects by mining data from mailing lists, issue trackers and versioning systems. Results indicate the potential usefulness of YODA as a recommendation system to aid project managers in supporting newcomers joining a software project.

Chapter 8 presents a recommender useful for code re-documentation. The usefulness of such a tool is motivated by the fact that, very often, source code lacks comments that adequately describe its behavior. In practice, the proposed approach mines messages exchanged among contributors/developers, in the form of bug reports and emails, and extract useful descriptions, that describe specific parts of the source code. We have evaluated the approach on bug reports and mailing lists from two open source systems (Lucene and Eclipse), and the obtained results suggest that the extracted method descriptions can help developers in understanding the code and could also be used as a starting point for source code re-documentation.

Chapter 7

Suggest Appropriate Mentors to help Newcomers in Open Source Projects

Contents

Program comprehension is crucial to apply each maintenance activity and thus, crucial for a newcomer to become active and terms of code changes. What a newcomer needs in such situation is a good summary of code that help him/her in comprehension and maintenance tasks. The previous Chapter motivates and reports an empirical study aimed at investigating to what extent a source code labeling based on IR techniques would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task. In recent years, researchers have applied various IR methods to ”label” software artifacts by means of some representative words. Such ”summaries” can help to improve the program comprehension of project newcomers that are join a software project. Thus, the key goal of this section try to improve such summaries of source code artifacts with the aim of facilitating their comprehension or just to better visualize them. These summaries can be useful source of information from project newcomers in understanding source code.

Thus, the extracted descriptions can help projects newcomer in understanding the code and could also be used as a starting point for source code re-documentation. However, when newcomers join a software project, they need to be properly trained to understand the technical and organizational aspects of the project. Clearly, the source code comprehension it is only a part of the technical skills that newcomer needs to properly acquire about a company. Inadequate training could likely lead to project delay or failure. Thus, a proper newcomer training and a good project environment are highly desirable because, can impact the probability of a newcomer to became a long term contributors [22]. The newcomers decision to abandon the project can be influenced by several factors [18, 22]. For example, as previous studies shown, there is a consequent low permanence rate of junior developers when they did not receive any answers (any support) by senior developers in the project [19, 22]. We argue that OSS projects, as well as, private Organizations need a proper newcomer training that is very often in the reality, organized and structured in a formal way a collocating in a formal training program called *Mentoring Program*. Let us to consider for example, for the Apache Software Foundation developers community, the page of the mentoring program [1] that explains formally what a mentor has to do in that company and what a mentee can expect from her mentor. In that page there is also the definition of Mentor: *A community member who makes a commitment to help a new contributor get started is a mentor*. Thus, not all committers are mentors and senior developers have the free choice to become mentor or not of project newcomers.

In this Chapter we propose an approach, named YODA (**Y**oung and newc**O**mer **D**eveloper **A**ssistant) aimed at identifying and recommending mentors in software projects by mining data from mailing lists and versioning systems. Candidate mentors are identified among

[1] https://community.apache.org/mentoringprogramme.html

experienced developers who actively interact with newcomers. Then, when a newcomer joins the project, YODA recommends her a mentor that, among the available ones, has already discussed topics relevant for the newcomer.

YODA has been evaluated on software repositories of 7 open source projects. We have also surveyed some developers of these projects to understand whether mentoring was actually performed in their projects, and asked them to evaluate the mentoring relations YODA identified. Results indicate that top committers are not always the most appropriate mentors, and show the potential usefulness of YODA as a recommendation system to aid project managers in supporting newcomers joining a software project. Finally, inspired by a work by Zhou *et al.* [22] we verify whether a newcomer that receive mentoring by experienced developers is more likely that she would stay with the project for a long time with respect to newcomers that did not receive support by mentors of the project. We find that a properly training by project mentors impacts the trajectory (career) of newcomers that are join the project. We discover that well trained newcomers have an higher permanence in the project, almost twice that of developers that do not receive any initial support.

7.1 Motivation: Who is Going to Mentor Newcomers in Open Source Project?

In the Star Wars imaginary Universe, YODA[2] is known to have trained a large number of young Jedi (*youngling*). Such a skill is equally important in software projects, where training new developers is a crucial activity. When a newcomer joins a project, she needs to be trained from many different points of view, such as project architecture and implementation details, development guidelines, and organizational aspects. Training is often performed by one or more *mentors* that, during the early stages of project participation for a newcomer, help her in the work and actively discuss with her project details. Once the newcomer has been trained, she can continue to work autonomously. A relatively similar process may occur in open source projects, where, in most cases, the interaction between the mentor(s) and the newcomer occurs through electronic means, e.g., mailing lists or issue tracking systems. A newcomer would likely first start participating to discussion actively, and then would gradually start to commit changes in the source code repository. Previous studies surveying software projects indicated that mentoring of project newcomers is highly desirable [25]. Differently from the problem of bug triaging—which requires to determine a developer who solved in the past a similar task [189]—mentoring requires to find developers having the ability to effectively communicate and train other people. Hence, *a good mentor is someone that (i) has enough expertise about the topic of interest for the newcomer, and that (ii) demonstrated to have enough ability to help other people.*

In this Chapter we propose YODA (**Y**oung and newc**O**mer **D**eveloper **A**ssistant) an approach to identify likely mentors in software projects by mining data from software repositories, and to support the project manager in recommending possible mentors when a newcomer joins a project. YODA is inspired by the *ArnetMiner*[3] search engine for academic researchers in computer science, which identifies relations between students and advisors, based on a series of heuristics [190]. Shortly, given a pair of researchers *Jim* and *Alice*, *ArnetMiner* suggests that *Jim* could have been the mentor of *Alice* based on four factors: f_1: *Jim* and *Alice* published together a large number of papers, f_2: *Jim* has more publications than *Alice*, f_3: *Jim* is older than *Alice* in terms of activity, and f_4: *Jim* and *Alice* started to publish together as soon as *Alice* started her activity. Even if the student-advisor relationship in academia is one-to-one, the metrics employed by *ArnetMiner* allow to also identify one-to-many relationships.

To identify candidate mentors in the past history of a software project, we consider as

[2] http://starwars.wikia.com/wiki/Yoda

[3] http://arnetminer.org

potential indicators on mentorship cases where (i) the mentor and the newcomer exchange a large number of emails in the first phases of the newcomer activity, (ii) the mentor has performed a larger activity in the project than the newcomer in terms of exchanged emails, (iii) the mentor has a longer experience in the project than the newcomer, (iv) the mentor and the newcomer start to exchange emails as soon as the newcomer joins the project, and (v) the mentor is very active in terms of performed commits. The set of indicators above defined is inspired to the *ArnetMiner*, although metrics being used are different: for example, co-authorship is replaced by level of communication; plus, differently from *ArnetMiner*, YODA needs to account both communication activity (measured in terms of exchanged emails) and technical activity (measured in terms of commits). Once being able to identify likely mentors in the past history of a software project, YODA recommends mentors each time a newcomer joins the project, by selecting, among the candidate mentors previously identified, those who discuss similar topics to what the newcomer is discussing.

We have evaluated YODA on data from seven open source projects, Apache httpd[4], Eclipse CDT[5], the FreeBSD kernel[6], JBoss Application Server[7] (in following we refear with JBossas), PostgreSQL[8], Python[9], and Samba[10]. First, we investigated which factors or combination of factors provide more accurate indicator of mentoring. Then, we evaluated the accuracy of YODA when recommending mentors for a newcomer. Finally, we also present the results of a preliminary survey conducted with some people involved in the suggested mentoring relationships of the seven projects to understand (i) whether what we identified was actually true, and (ii) to understand how mentoring was performed (and if it was performed) in that project. Moreover, inspired by a work by Zhou *et al.* [22] we verify whether a newcomer that receive mentoring by experienced developers is more likely that she would stay with the project for a long time with respect to newcomers that did not receive any support. We also implemented our approach in an Eclipse plugin, named YODA (available at the `http://www.ing.unisannio.it/spanichella/pages/projects.html`), that identifies and recommends mentors for newcomers joining a software project by mining developers' communication (e.g., mailing lists) and project versioning systems.

The Chapter is organized as follows. Section 7.2 describes YODA, the approach proposed to identify and recommend mentors. Section 7.3 describes the empirical study aimed at evaluating YODA on the seven open source projects. Section 7.4 reports the study results.

[4] `http://httpd.apache.org/`
[5] `https://www.eclipse.org/cdt/âĂŐ`
[6] `www.freebsd.org/`
[7] `https://www.jboss.org/`
[8] `www.postgresql.org/âĂŐ`
[9] `www.python.org/`
[10] `https://www.samba.org/`

Section 7.5 provide further specific results together with a qualitative discussion of the empirical study performed, while Section 7.6 discusses the limitations and threats to validity of both YODA and of the study. Section 7.7 describes the Eclipse plugin YODA. Finally, Section 7.9 concludes the results of the study after a discussion of the related literature (Section 7.8).

7.2 How to Identify Mentors

To recommend mentors, YODA first identifies a list of available mentors by mining mailing lists (and both issue trackers and mailing lists) and versioning systems. Then, it analyzes the requests for help issued by a newcomer and identifies a list of candidate mentors that could help her. The next sections detail the approach we use to (i) identify mentors from historical data of existing projects; and (ii) to recommend mentors when a newcomer joins a project.

7.2.1 Who could be a good mentor?

To identify mentors in software projects, we define factors inspired by the ones *ArnetMiner* uses. However, we replace the activity of co-authoring a paper with the activity of exchanging emails and alternatively with the activity of exchanging emails together with the activity of exchanging messages in issue trackers discussion. This, means that we have each *social* metric inspired by ArnetMiner two versions: a first version that consider only (i) the social activity of developers in mailing lists and a latter that considers both (ii) the social activity of developers in mailing lists and issue trackers. In addition, we consider the technical activity of a candidate mentor, in terms of number of commits. Specifically:

- f_1: captures whether, after a newcomer joins a project, she mainly collaborates with a specific person. We measure this as the percentage of emails a newcomer exchanges with an older project member in the first period after joining the project, out of the total number of emails exchanged in that period. Alternatively, we measure this as the percentage of emails plus the percentage of messages in the issue tracker that a newcomer exchanges with an older project member, out of the total number of emails plus the number messages in issue trackers exchanged in that period.

- f_2: captures the difference of the amount of discussion activity carried out by mentors and newcomers in terms of the number of emails exchanged. Alternatively, we measure this as the difference of the amount of discussion activity carried out by mentors and newcomers considering both number of emails in the mailing list and the messages in issue trackers.

- f_3: captures the "age" difference between the newcomer and the likely mentor, i.e., the difference (in months) between the date when the newcomer exchanged her first email (or the first message in the issue tracker if the date is earlier) in the project and the date when the likely mentor did.

- f_4: tells whether the mentor is one of the people the newcomer starts to collaborate first. It is defined in terms of time difference (in months) between the first email exchanged by a newcomer (or the first message in the issue tracker if the date is earlier), and the first email (or the first message in the issue tracker if the date is earlier) the newcomer exchanged with the likely mentor.

- f_5: considers the number of commits performed by a candidate mentor, as an indicator of the past *technical* activity performed by the candidate mentor.

To allow aggregating the five factors, we normalize them in the interval $[0 \dots 1]$, and then aggregate them into a score defined as

$$\sum_{i=1}^{5} w_i \cdot f_i \tag{7.1}$$

where w_i is the importance (weight) attributed to f_i. Determining appropriate weights w_i for f_i—and, as a consequence, also whether each of the f_i is considered or not—is part of the empirical evaluation and will be detailed in Section 7.3.1. It is important to note that we aggregating the five factors as above described, considering mailing list data only or alternatively considering both mailing list and issue tracker data. This means that we have two versions of the score, a first version that aggregating the five factors obtained considering only the mails data and a latter version of the score that that considers both mails and issue data.

By using the score defined above, it is possible—from the past history of a project—to rank all other developers with the aim of identifying the list of available mentors that can be suggested to newcomers. The higher the rank, the higher the likelihood that the developer played the role of mentor for the considered newcomer. Once the developers have been ranked, a threshold is used to cut the ranked list and identify the top related developers that represent the candidate mentors.

Defining a "good" threshold *a priori* to cut the ranked list is challenging, because a newcomer could have one or more mentors. Thus, we use a scaled threshold t—used in previous work on traceability recovery [169]—based on the values of the factors computed considering the newcomer i and the top developers in the ranked list $t_i = \lambda \cdot TOP_i$, where TOP_i is the value of the factor between the newcomer i and the top developer in the ranked list, while

Table 7.1: Characteristics of the five projects analyzed, and of the training and test sets for evaluating YODA.

VARIABLE	APACHE HTTPD	ECLIPSE CDT	FREEBSD	JBOSSAS	POSTGRESQL	PYTHON	SAMBA
Period	08/2001-	06/2002-	11/1998-	06/2001-	10/1998-	05/2000-	04/1998-
considered	12/2008	06/2012	10/2008	06/2012	03/2008	12/2008	12/2008
Size range (KNLOC)	271-850	45-1,528	3,552-7,853	642-1,272	223-522	464-683	156-1,157
Mailing list contribs (Mc)	6,726	89	23,872	112	2,935	20,827	3,411
Issue Tracker contribs (Ic)	5,769	2,179	2,646	2,759	5,628	326	4,984
Committers (Cm)	108	91	640	337	34	147	229
$Mc \cap Cm$	66	87	393	105	29	147	226
$Ic \cap Cm$	11	63	35	72	28	62	127
$(Mc \cup Ic) \cap Cm$	66	90	398	134	30	147	227
Emails exchanged by $Mc \cap Cm$	135,243	5,138	2,246,425	6336	44,244	7,479	19,073
Bugs reported/discussed by $Ic \cap Cm$	2,498	9,193	9,964	96,7774	7,190	3,601	22,238
Period of	08/2001-	06/2002-	11/1998-	06/2001-	10/1998-	05/2000-	04/1998-
the Training Set	03/2002	06/2005	02/2000	06/2004	05/2001	05/2001	09/2000
Period of	04/2002-	07/2005-	03/2000-	07/2004-	06/2001-	06/2001-	10/2000-
the Test Set	12/2008	06/2012	10/2008	06/2012-	03/2008	12/2008	12/2008
Training/Test Set: Mails only							
# of mentors (training set)	18	42	81	42	17	28	31
# of newcomers (training set)	11	31	30	34	8	28	31
# of newcomers (test set)	11	31	30	34	7	28	30
Training/Test Set: Mails and Issues							
# of mentors (training set)	27	44	87	73	22	68	73
# of newcomers (training set)	11	31	75	53	11	38	91
# of newcomers (test set)	11	31	75	53	10	37	90

$\lambda \in [0, 1]$. The defined threshold is used to remove from the ranked list developers that have a factor value lower than $\lambda\%$ of the factor value between the newcomer and the top ranked developer.

7.2.2 Building the project committers' communication network

We identify communication between developers by analyzing mailing lists and issue trackers and by linking names/email in in such sources to committer IDs extracted from the versioning repository. The link to versioning repositories is needed since we are interested to select people who committed changes to the project repository.

Unifying Project Contributors' Names

We use an approach similar to the one used by Bird *et al.* [27] and the one use by Canfora *et al.* [45] as described in Section 2.2.2.

Extracting Developers' Links

Once unified the names, **we restrict our attention to commit authors'** only. This is because we want to focus our attention to discussions occurring between people involved in code changes only, rather than other people participating to the discussions.

Given two project's members, M_i and M_j, we identify a link $M_i \leftrightarrow M_j$ between them in the different sources of information by applying the same heuristics used in Section 2.2.2.

7.2.3 Recommending mentors

When a contributor joins a project, she needs to become knowledgeable on a specific (set of) topic(s). The project manager needs to find a mentor for this newcomer, that is a contributor that (a) demonstrated already to be a mentor in the past, and (b) actually worked/discussed topics related to those the newcomer is going to work. As for requirement (a), we use the method defined in Section 7.2.1 to identify available mentors in the past project history. Turning to requirement (b), we need to select among the available contributors that demonstrated to be mentors in the past, the ones that had enough expertise on the topic(s) which the newcomer is going to work on. Rather than proposing a novel approach to identify the most appropriate mentor for a given topic, we use approaches similar to what proposed in the past for bug triaging [189, 191, 192], which proposed to assign a bug to a developer that in the past worked on a bug having a (textually) similar bug report. Specifically, let us suppose a newcomer p joins the project at time t_x, and let us consider the set of n mentors $M = \{m_1, \dots m_n\}$ identified using the method described in Section 7.2.1 in the period before t_x. We instantiate an IR [80] process to rank the available mentors, where each document d_i with $i = 1 \dots n$ consists of the union of the text of all emails exchanged by the mentor m_i before t_x, while the query q_p is represented by a request for help submitted by the newcomer p. Alternatively, we instantiate a similar, IR process, where the only difference is that we that consider as document, the union of the text of all emails together with issue messages exchanged by the mentor m_i before t_x.

In a live setting—when YODA would be instantiated as a tool—the request for help can be explicitly submitted by the newcomer when she joins the project. For the purpose of our empirical study, since we are working on past data from open source projects, we simulated the query by considering the union of the emails (or both emails and issue trackers messages) sent by the newcomer during the first week when she joined the project, i.e., within one week after the first email (or the first message in the issue tracker if the date is earlier). We consider a week as a time span on the one hand long enough for a newcomer to send emails (or issue tracker message) and precisely ask what kind of help she needs (e.g., taking the first email only could be considered too vague), on the other hand not so long to have an unrealistic scenario.

Both documents and query are processed by removing English stop words, performing stemming using Porter stemming [156], and indexing the text corpus using word frequencies. We use raw frequencies (*tf*) [80] rather than the widely used *tf-idf* [80] as our aim is to mach similar corpus rather than to discriminate different documents, reason for what *tf-idf* results useful. Then, we compare the query q_p with each d_i using the asymmetric Dice

coefficient [80]:

$$Dice_p = \frac{\sum_{j=1}^{N} tf_{j_{q_p}} \cdot tf_{j_{d_i}}}{\sum_{j=1}^{N} tf_{j_{q_p}} \cdot tf_{j_{q_p}}}$$

where N is the size of the vocabulary of all words contained in our documents, $tf_{j_{q_p}}$ and $tf_{j_{d_i}}$ are the raw frequencies of the j^{th} dictionary term in the query and d_i, respectively. In other words, the asymmetric Dice coefficient captures how much text of the newcomer's request for help is covered by the mentor's emails (and issue tracker messages). We use it instead of the *cosine* coefficient because the Dice coefficient does not penalize mentors having email corpus that contains much text not contained in the newcomer's request.

7.3 Empirical Study Definition

The *goal* of this study is to evaluate the performances of YODA in identifying mentors in software projects, and in recommending mentors for project newcomers. The *quality focus* is concerned with the capability to precisely identify and recommend mentors that (i) had a mentoring experience in the past and (ii) have discussed topics related to the newcomer needs. The *context* consists of discussions extracted from mailing lists/issue trackers and changes extracted from versioning systems for seven open source projects, namely Apache httpd, Eclipse CDT, the FreeBSD kernel, JBoss, PostgreSQL, Python, and Samba.

The seven projects are different in terms of size and application domain. Apache httpd is an open-source HTTP server, Eclipse CDT is a C/C++ Integrated Development Environment, FreeBSD is a free, open source Unix operating system, JBossas is an open source application server written in Java, PostgreSQL is a database management system (DBMS), Python is a well-known scripting language, and Samba a cross-operating system layer for printer and file sharing. The top part of Table 7.1 reports key information about the seven projects and, above all, of their mailing lists and versioning systems, namely project URL, time period analyzed, size range in KLOC, number of mailing list contributors (Mc), number of committers in the versioning systems (Cm), their intersection ($Mc \cap Cm$), and number of emails exchanged by $Mc \cap Cm$, i.e., by the set of project contributors we considered. In addition, about the issue tracker data, the table reports also the number of issue tracker contributors (Ic), the intersection with committers in the versioning systems ($Mc \cap Cm$) and number of messages exchanged by $Ic \cap Cm$.

7.3.1 Study Procedure

The study addresses the following research questions:

- **RQ$_1$:** *How can we identify mentors from the past history of a software project?* This question aims at investigating whether particular combinations of the factors f_1–f_5 described in Section 7.2.1 can be used to identify mentors, and how good are such combinations compared with some baselines, e.g., using as mentors the project managers or the top project contributors.

- **RQ$_2$:** *To what extent would it be possible to recommend mentors to newcomers joining a software project?* This research question investigates how accurately could YODA recommend a mentor—among those identified in **RQ1**—for a newcomer that joins the project and is willing to work on a certain topic.

- **RQ$_3$:** *How does mentoring activity affects the future trajectory of a newcomer when she joins the project?* This research question investigates and compares the trajectory of newcomers that had at least a mentor—among those identified in **RQ$_1$**— in their past with newcomers that did not receive support by any mentors. We want to verify whether a newcomer that receives mentoring by experienced developers is more likely that she would stay with the project for a long time.

To address **RQ$_1$**, we use different combinations of f_1–f_5 to identify—for each newcomer joining a project—a ranked list of candidate mentors, and then we evaluate them manually. It is worthwhile to recall that such a metrics combination suggests a ranked list of candidate mentors for each newcomer by observing the past project history, without however requiring a training set (the approach is unsupervised) and that, in the context of **RQ$_1$** we are only interested to see whether the f_1–f_5 metrics can identify good mentors, without checking whether the mentor has appropriate skills required by the newcomer (we deal with such an issue in **RQ$_2$**).

First, we perform a calibration of the weights in equation (7.1). We consider different possible combinations of f_1–f_5, i.e., all ones with equal weight, single factors alone, all possible pairs, all groups of three, and four factors. It is important to note that, by considering f_5 (normalized number of commits) alone, we provide a sort of baseline for our technique, because the number of commits is a metric often used to identify experts—or at least code ownership [193]—in software projects [194].

Then, we give varying weights to the five factors, with the aim of investigating whether there are factors that are more important than others. For example, for combinations of three factors, say f_1, f_2, f_3, we consider, other than the combination with equal weights, i.e., $0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$, the following cases: (i) one factor weighted more than the others, e.g., $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$, and similarly for f_2 and f_3; and (ii) two factors weighted more than the other, e.g., $0.4 \cdot f_1 + 0.4 \cdot f_2 + 0.2 \cdot f_3$, and similarly for (f_1, f_3) or

(f_2, f_3).

As specified in Section 7.2, we applying the analysis described above varying the weights of the five factors using in a first case (i) mailing list data only and alternatively considering (ii) both mailing list and issue tracker data. We apply this ulterior analysis to verify whether the results change if considering the two sources of information together.

To validate the candidate newcomer-mentor pairs, one of the authors and another PhD student (not aware of how YODA works) manually inspected (independently, discussing cases where they disagreed) the communication between the newcomer and the mentor. The pair mentor-newcomer is classified as a *true positive* if there is a clear evidence of cases in which the newcomer asked help to—and received help from—the mentor, otherwise it is classified as a *false positive*.

To address $\mathbf{RQ}_2$, we split the project history into two periods, a first period (*training set*) in which we identify the set M of people who have been identified as candidate mentors using the most suitable combination of factors determined in $\mathbf{RQ}_1$, and a second period (*test set*) for which we aim at recommending mentors for project newcomers. For each project, we have chosen training and test set in order to have a balanced number of newcomers in the two sets (see the bottom part of Table 7.1). It is important to note that we used in a first case (i) mailing lists only, and in a second case (ii) both mailing lists and issue trackers to address such research questions. Thus, we have two test sets and two training sets as reported in Table 7.1.

For each newcomer i identified in the test set, we identify the top k mentors ($m_j \in M$) that exhibit the most similar discussion to the newcomer request for help. For each possible newcomer, we produce a ranked list of k recommended mentors, as in realistic scenarios it can happen that the most suitable mentor is not available, and one has to choose the second-best, the third-best, and so on. Since in the context of a post-mortem analysis of data from open source projects like the one we are doing we do not have available other sources of requests for help, we assume that the topic on which a newcomer requires helps is contained in the first emails she exchanges in the project. We define below, in the study settings, how the number of emails (or issue tracker messages) has been chosen. Then, for each newcomer i in the test set, we identify mentor(s) using the approach described in Section 7.2.3, manually validate it, compare with the recommended mentor, and report the number and percentage of pairs for which the recommendation was correct or incorrect.

To address $\mathbf{RQ}_3$, we split the set of projects newcomers into two subsets, a first subset in which there are the set N_m of people who had a mentor(s) when they joined the project and a second subset in which contains N_a of people that did not receive support by mentors. Specifically, similarly to the work by Zhou *et al.* we consider a developer as a Long Term

Contributor (**LCT!**) if she contributed to the project for at least 28 or more months. We use such information to measure the longevity of the life of developers in the project. This means that the first and the last date that a newcomer contributes to a project represent the timing margins of her life in a project. To answer this research question we relying on both mailing lists and issue trackers as sources of information. Consequently, we consider as developer contribution (i) a change in the code, (ii) an email in the mailing list and (iii) a message in the issue tracker. The first and last of these contributions of the developer constitute the temporal margins of her time of permanence in the project. Moreover, to give a more precise quantitative measure of the probability to become a LCT when a developer receive or not mentoring (by experienced developers) we also computed the odds ratio (OR) [195] that indicates how much an event to occur or not. In epidemiology, the odds ratio (OR) is one of the indices used to define the relationship of cause and effect between two factors, for example, between a risk factor and a disease. In such scenario, the calculation of odds ratio includes the comparison between the frequencies of occurrence of the event (eg, disease) in subjects exposed and those not exposed to the risk factor in the study, respectively. Formally, in our context, the odds ratio is defined as the ratio of the odds p of an event occurring in one sample, i.e., the set of newcomers that had a mentor and become LCTs (experimental group), to the odds q of it occuring in the other sample, i.e., the set of newcomers that had not a mentor and become LCTs (control group):

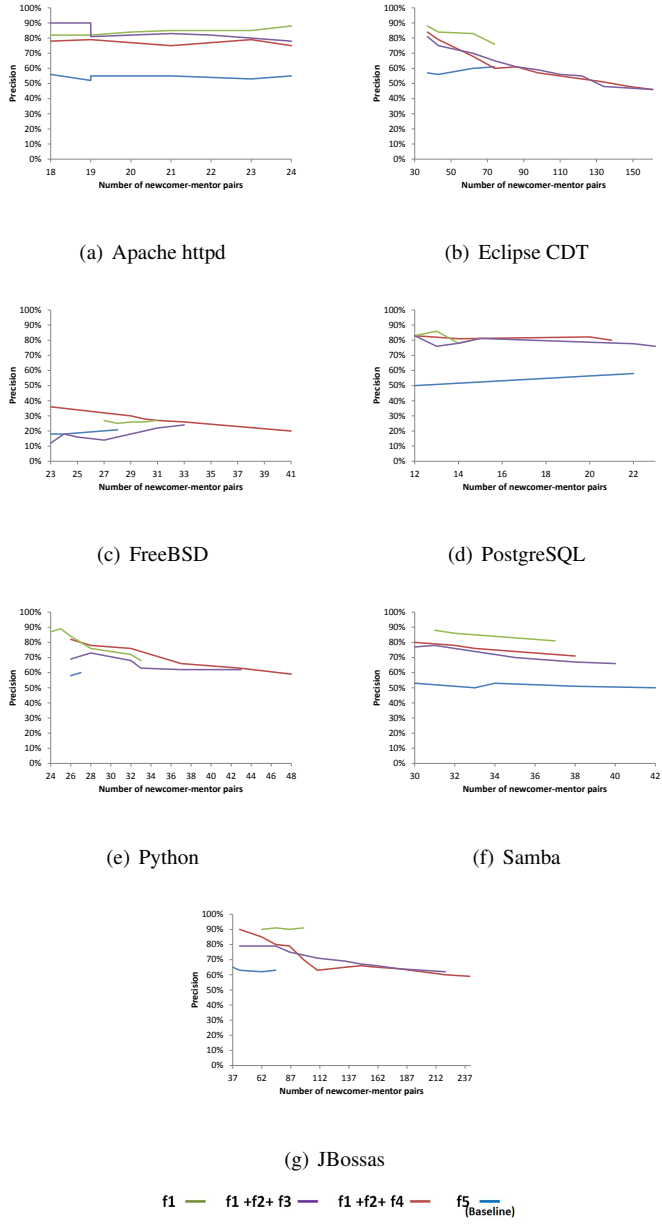
$$RO = \frac{p/(1-p)}{q/(1-q)}$$

where an *OR* greater than 1 means that the event is more likely in first sample (newcomers that had a mentor). Vice versa, an *OR* less than 1 indicates that the event is more likely in the second sample (newcomers no mentored) while, an *OR* equal to 1 indicates that the event is equally likely in both the samples.

7.3.2 Surveying project developers

As a further evaluation, after having identified candidate pairs of mentor-newcomer, we surveyed these developers by sending them an email, explaining them the purpose of our research activity, and asking them to complete an on-line survey questionnaire—built using *SurveyMonkey*[11]—asking questions about: (i) the experience as project contributor, and whether the contributor is still active; (ii) whether the person mentored a project newcomer and, if yes, how important was the mentoring perceived; (iii) whether the person was mentored when she joined the project, and how important was the mentoring in the decision to

[11] www.surveymonkey.com



(a) Apache httpd

(b) Eclipse CDT

(c) FreeBSD

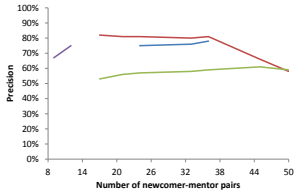
(d) PostgreSQL

(e) Python

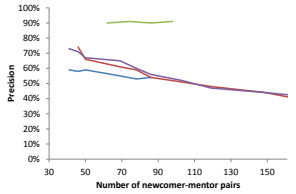
(f) Samba

(g) JBossas

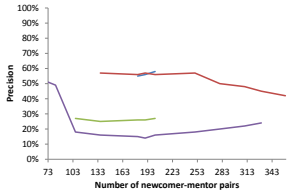
Figure 7.1: Mentor identification performances for the best combinations of f_1–f_5 and for the baseline (f_5) considering mailing list data



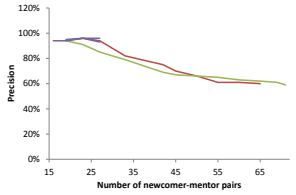
(a) Apache httpd



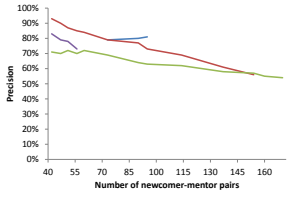
(b) Eclipse CDT



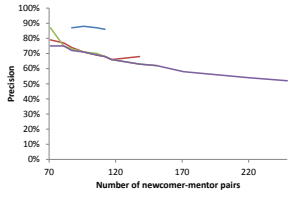
(c) FreeBSD



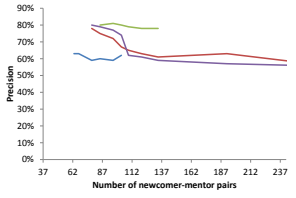
(d) PostgreSQL



(e) Python



(f) Samba



(g) JBossas

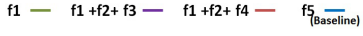


Figure 7.2: Mentor identification performances for the best combinations of f_1–f_5 and for the baseline (f_5) considering mailing list and issue tracker data

stay in the project; and (iv) which are the characteristics of a good mentor (e.g., experience, communication skills, project knowledge).

In addition, the survey page showed to respondents a list of people that YODA highlighted in the mentor-newcomer candidate pairs, asking them to tell, for each person, whether the respondent (i) was actually the person indicated there, (ii) was a mentor for that person, (iii) was advised by that person, or (iv) never got in touch with that person. We invited to the survey 23 developers from Apache httpd, 19 developers from Eclipse CDT, 37 from FreeBSD, 20 developers from JBossas, 15 from PostgreSQL, 27 from Python, and 37 from Samba. The questionnaire was completed by 6 developers from Samba, 3 from FreeBSD, 3 from JBoss, 2 from PostgreSQL, 2 from Eclipse CDT and 1 from Python.

7.4 Results

This section reports the results of the empirical study.

7.4.1 RQ_1: How can we identify mentors from the past history of a software project?

Figure 7.1 and Table 7.2 report, for the seven projects we analyzed, the performances of YODA in detecting mentors for all newcomers joining the project during the entire analyzed time interval considering mailing list (only). In each subfigure of Figure 7.1, the x-axis indicates the number of mentors that the approach can recommend when achieving a precision shown on the y-axis. Vice versa, Figure 7.2 (contain the same information of Figure 7.1) and Table 7.3 report, for the seven projects we analyzed, the performances of YODA in detecting mentors for all newcomers joining the project during the entire analyzed time interval considering both mailing list and issue tracker. As described in Section 7.3.1, we evaluated the performance of YODA for all combinations of f_1–f_5, as well as for combinations having varying weights. However, Table 7.2 and Table 7.3 only report a subset of the analyzed combinations.

Specifically in Table 7.2:

- f_1, i.e., the percentage of exchanged emails between newcomer and mentor. We noticed that, if considering each of the f_1–f_5 factors alone, such a factor is the one that produces the best performances.
- f_5, i.e., the normalized number of commits performed by the candidate mentor. We use this factor as a baseline, to determine whether an obvious choice of mentors, i.e., top committers, could be appropriate.

Table 7.2: Precision (%) and number of newcomer-mentor pairs identified for different values of λ considering only Mailing lists data.

System	Formula	$\lambda = 100$		$\lambda = 90$		$\lambda = 80$		$\lambda = 70$		$\lambda = 60$		$\lambda = 50$	
Apache httpd	f_1	83%	18	83%	18	**83%**	18	**84%**	19	**84%**	19	**86%**	21
	$f_5(baseline)$	56%	18	53%	19	55%	20	55%	20	52%	21	54%	26
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	83%	18	85%	20	82%	22	83%	24	80%	25	77%	26
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	78%	18	75%	20	77%	22	77%	22	77%	22	78%	23
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	**89%**	18	**89%**	18	81%	21	83%	23	79%	24	77%	26
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	78%	18	79%	19	79%	19	76%	21	78%	23	75%	24
Eclipse CDT	f_1	88%	43	**88%**	43	**85%**	46	**84%**	49	**83%**	60	**78%**	72
	$f_5(baseline)$	57%	37	56%	43	57%	44	60%	47	59%	54	61%	75
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	81%	37	75%	53	65%	75	55%	103	48%	136	46%	168
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	84%	37	75%	52	68%	65	58%	93	51%	134	46%	161
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	**92%**	37	83%	54	66%	74	63%	87	56%	110	49%	148
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	89%	37	83%	53	72%	65	62%	82	56%	109	51%	140
FreeBSD	f_1	26%	27	26%	27	**26%**	27	**26%**	27	**25%**	28	**26%**	31
	$f_5(baseline)$	17%	23	17%	23	17%	23	17%	24	17%	24	21%	29
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	17%	23	17%	24	17%	24	17%	24	19%	26	23%	31
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	30%	23	25%	28	20%	35	20%	35	21%	39	20%	40
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	13%	23	13%	23	17%	24	15%	27	20%	30	24%	33
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	**35%**	23	**31%**	26	24%	33	22%	37	21%	38	20%	41
JBossas	f_1	**90%**	63	90%	63	**91%**	67	**92%**	73	**90%**	89	**91%**	97
	$f_5(baseline)$	65%	37	62%	39	62%	45	63%	54	62%	63	63%	71
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	79%	43	78%	76	72%	92	69%	137	63%	189	61%	221
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	89%	46	82%	83	66%	152	64%	194	60%	220	59%	241
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	81%	43	82%	68	82%	98	75%	130	65%	181	62%	211
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	89%	46	**92%**	71	80%	101	69%	153	65%	192	62%	221
PostgreSQL	f_1	**83%**	12	**85%**	13	**79%**	14	**79%**	14	**80%**	15	**80%**	15
	$f_5(baseline)$	50%	12	50%	12	50%	12	56%	16	62%	21	64%	22
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	**83%**	12	76%	17	75%	20	65%	23	62%	26	55%	29
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	**83%**	12	77%	13	73%	15	76%	21	73%	22	70%	23
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	**83%**	12	77%	13	79%	14	75%	20	73%	22	70%	23
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	**83%**	12	79%	14	79%	14	**79%**	14	**80%**	20	76%	21
Python	f_1	**88%**	24	**88%**	24	**88%**	25	**82%**	28	**74%**	31	**70%**	33
	$f_5(baseline)$	58%	26	58%	26	59%	27	59%	27	59%	27	59%	27
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	62%	26	68%	31	66%	32	64%	36	63%	40	61%	41
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	73%	26	75%	28	74%	34	63%	40	60%	45	57%	47
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	69%	26	71%	28	66%	32	64%	33	62%	39	60%	43
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	81%	26	81%	26	75%	32	65%	37	62%	42	58%	48
Samba	f_1	**87%**	31	**84%**	32	**84%**	32	**80%**	35	**80%**	35	**78%**	37
	$f_5(baseline)$	53%	30	52%	33	52%	33	53%	34	53%	36	50%	42
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	77%	30	72%	32	69%	35	69%	36	66%	38	64%	42
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	73%	30	70%	33	68%	34	67%	36	63%	40	63%	41
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	77%	30	77%	31	69%	35	69%	35	68%	37	65%	40
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	80%	30	78%	32	78%	32	76%	33	72%	36	71%	38

- $0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$ and $0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$, because we considered that the combination of f_1 and f_2 with either f_3 or f_4 produces the best results among combinations obtained weighting all factors similarly.
- $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$ and $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$, i.e., again considering f_1 and f_2 with f_3 or f_4, weighting f_1 twice than the other factors. These are the combinations able to achieve the best performances.

Vice versa, in Table 7.3:

- f_1, i.e., the percentage of both exchanged emails and messages in the issue trackers

Table 7.3: Precision (%) and number of newcomer-mentor pairs identified for different values of λ considering both Mailing lists and Issue Trackers data.

System	Formula	$\lambda = 100$		$\lambda = 90$		$\lambda = 80$		$\lambda = 70$		$\lambda = 60$		$\lambda = 50$	
Apache httpd	f_1	75%	24	75%	24	75%	24	75%	24	**77%**	26	**78%**	36
	$f_5(baseline)$	67%	9	70%	10	73%	11	75%	12	75%	12	75%	12
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	53%	17	57%	23	56%	32	58%	40	61%	46	59%	49
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	82%	17	77%	30	**81%**	36	66%	44	60%	48	58%	50
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	76%	17	**80%**	20	69%	26	62%	37	63%	46	63%	46
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	**82%**	17	78%	23	78%	32	**81%**	36	71%	41	63%	46
Eclipse CDT	f_1	59%	71	59%	71	**59%**	71	59%	78	**60%**	86	**61%**	88
	$f_5(baseline)$	59%	41	58%	45	**59%**	49	**60%**	50	55%	69	55%	82
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	73%	41	64%	70	52%	103	44%	149	40%	179	38%	204
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	74%	46	61%	69	48%	119	40%	166	38%	199	37%	217
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	**80%**	41	**70%**	57	**59%**	85	53%	112	44%	154	40%	189
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	74%	46	62%	65	58%	83	50%	122	42%	166	40%	194
Freebsd	f_1	55%	181	55%	181	**55%**	181	**56%**	188	**58%**	198	**58%**	202
	$f_5(baseline)$	51%	73	50%	74	49%	76	49%	79	49%	80	49%	82
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	55%	105	49%	141	48%	192	48%	250	44%	304	42%	339
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	57%	108	57%	208	48%	285	42%	343	41%	355	41%	359
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	57%	105	54%	149	53%	195	50%	247	45%	307	42%	345
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	**58%**	108	**59%**	185	**55%**	229	47%	289	43%	343	41%	356
JBossas	f_1	80%	86	**81%**	89	**81%**	96	**80%**	105	**78%**	120	**78%**	134
	$f_5(baseline)$	67%	63	63%	67	59%	78	60%	84	60%	97	62%	102
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	80%	74	76%	104	66%	108	61%	145	54%	290	52%	347
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	78%	74	77%	114	63%	192	58%	251	54%	314	50%	369
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	75%	73	75%	103	73%	133	67%	188	60%	257	56%	299
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	**84%**	74	77%	99	70%	141	65%	199	58%	263	54%	311
PostgreSQL	f_1	**95%**	19	95%	20	95%	22	**96%**	23	**96%**	25	**96%**	26
	$f_5(baseline)$	94%	16	94%	18	95%	20	**96%**	23	92%	26	93%	27
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	94%	18	91%	23	79%	33	69%	42	64%	53	59%	71
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	94%	18	**96%**	24	**96%**	25	81%	37	61%	56	60%	65
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	94%	18	91%	23	92%	26	84%	31	74%	42	67%	51
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	94%	18	95%	21	**96%**	25	**96%**	26	88%	34	65%	54
Python	f_1	79%	73	80%	74	80%	74	**80%**	79	**80%**	91	**81%**	95
	$f_5(baseline)$	83%	42	80%	45	79%	47	79%	48	78%	51	73%	56
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	71%	42	72%	60	64%	90	62%	114	58%	137	54%	170
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	**93%**	42	**85%**	55	**84%**	63	77%	88	69%	111	56%	154
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	81%	42	81%	58	76%	75	70%	99	63%	133	57%	169
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	91%	84	82%	73	72%	97	68%	130	56%	190	47%	238
Samba	f_1	87%	87	**87%**	87	**88%**	90	**88%**	96	**87%**	106	**86%**	112
	$f_5(baseline)$	75%	71	75%	81	71%	96	66%	117	63%	138	62%	151
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$	87%	71	74%	97	64%	135	58%	171	54%	220	52%	249
	$0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$	79%	71	77%	82	74%	94	69%	107	68%	118	68%	139
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$	**92%**	71	86%	84	80%	99	69%	136	62%	163	56%	234
	$0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$	82%	71	80%	83	79%	92	78%	103	76%	118	69%	145

between newcomer and mentor. We noticed that also in this case, similarly with what we found in Table 7.2, if considering each of the f_1–f_5 factors alone, such a factor is the one that produces the best performances. However, with respect to the results in Table 7.2, in Table 7.3 the difference betweens the results of f_1 and f_5 (that is the baseline) is lower.

- f_5, i.e., the normalized number of commits performed by the candidate mentor. We use this factor as a baseline, to determine whether an obvious choice of mentors, i.e., top committers, could be appropriate.

- $0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_3$ and $0.33 \cdot f_1 + 0.33 \cdot f_2 + 0.33 \cdot f_4$, because we considered

that the combination of f_1 and f_2 with either f_3 or f_4 produces the best results among combinations obtained weighting all factors similarly (as in Table 7.2).

- $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$ and $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$, i.e., again considering f_1 and f_2 with f_3 or f_4, weighting f_1 twice than the other factors. These are the combinations able to achieve the best performances (as in Table 7.2).

Note that Figure 7.1 (and Figure 7.2) contains exactly the same information of Table 7.2 (except that we do not report the combination of f_1, f_2, and f_3 or f_4 with equal weights since they are less interesting to be compared), and has the purpose of better allowing the comparison of different combinations, whereas Table 7.2 (and Table 7.3) provides precise figures.

In summary, the obtained results indicate that:

- f_5 (number of commits) is not a very good indicator of mentorship (especially if we consider data from mailing lists only). Also, f_5 does not even provide a useful contribution if used in combination with other factors. While it is true that a very active committer can be expert on a particular topic, she might not be very willing (or able) to exchange ideas and/or instruct other people. It is important to highlight this finding for f_5 vary if consider data from both mailing lists and issue trackers. Indeed, in average the precision of such a factor improve for all the projects. However, results achieved relying on f_5 are not, in general, better than the results achieved relying on social factors like f_1.

- $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$ and $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$ achieve the best performances, with a precision over 70% and in many cases well above 80%. Also, it can be noticed that such a precision does not decrease when decreasing λ, which means that the chosen combinations are able to recommend a wider set of candidate mentors without sacrificing the precision.

- f_1 alone tend to be as precise as the other combinations, and in some cases (Python and Samba) even more precise than the combinations above. However, it is only able to recommend a limited set of mentors even when decreasing λ. Although, in the practice, a single mentor would suffice, having a good recall is desirable because (i) not all candidate mentors may be available when needed, and (ii) among the candidate mentors, we need to select those having the expertise required by the newcomer. Hence, to obtain a good balance between precision and number of mentors (which as said above does not mean to accept a very low precision), it is necessary to combine f_1 (having however a higher weight) with f_2 (rewarding mentors in general very active

in discussions) and f_3 (mentor/newcomer project age difference) or f_4 (the newcomer mainly collaborate with the mentor in her early stages). In summary, the percentage of exchanged emails (and messages in issue trackers) *per se* is not enough to identify a large set of candidate mentors.

- results in Table 7.2 and Table 7.3 are in general consistent with each other. However, the number of newcomer-mentor pairs that we can achieve is higher in Table 7.3 with respect to Table 7.2. This means that considering both mailing lists and issue trackers impact positively the precision in the results of YODA; in addition, the number of identified mentors (in general also the total number of developers and newcomers) is higher when we consider all of these sources of information. Thus, the use of more than one source of information helps to improve both *recall* (number of correct mentors identified) and *precision* in mentors identification.

The results achieved also indicate that on FreeBSD YODA exhibits the lower performances. Specifically, the precision is around 30% in Table 7.2 and 55% in Table 7.3. In this case it is interesting to find that using both mailing lists issue trackers data we can mitigate the very low performances in terms of precision that we obtain considering mailing lists data only. For all other systems results are pretty higher and consistent. Our interpretation is that such a result does not depend on the system size and on its number of project contributor. In fact, on other systems having a high number of contributors (such as JBossas, Python or Samba), YODA exhibits good performances. We believe that this can be due to the nature of the discussion occurring in the FreeBSD mailing lists. By inspecting the emails we realized that most of the discussion is one-to-many, i.e., people posting issues or suggestions to many other people or to the whole mailing list. Although the way we analyze mailing lists allow to treat—with some approximation many-to-many communications—the FreeBSD communication did not always exhibit dominant persons (potential mentors). This, clearly, makes the identification of pairwise collaborations less precise.

RQ$_1$ summary: *It is possible to identify mentors—with a precision of 70% or above—in the past history of a software project by using the combinations of factors* $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$ *or* $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_4$. *Considering top committers is not a precise and reliable metric to identify mentors.*

Table 7.4: Number and percentage of correct and incorrect top 1 and top 2 mentor recommendations for newcomers in the test set considering only Mailing lists.

	TOP 1		TOP 2	
SYSTEM	Correct	Wrong	Correct	Wrong
Apache	11 (85%)	2 (15%)	21 (81%)	5 (19%)
Eclips CDT	16 (80%)	4 (20%)	29 (76%)	9 (24%)
FreeBSD	10 (30%)	23 (70%)	16 (24%)	50 (76%)
JBossas	21 (95%)	1 (5%)	41 (95%)	2 (95%)
PostgreSQL	7 (100%)	0 (0%)	14 (100%)	0 (0%)
Python	20 (65%)	11 (35%)	48 (77%)	14 (23%)
Samba	31 (94%)	2 (6%)	54 (82%)	12 (18%)

Table 7.5: Number and percentage of correct and incorrect top 1 and top 2 mentor recommendations for newcomers in the test set considering both Mailing lists and Issue Trackers.

	TOP 1		TOP 2	
SYSTEM	Correct	Wrong	Correct	Wrong
Apache	8 (80%)	2 (20%)	13 (72%)	5 (28%)
Eclips CDT	19 (86%)	3 (14%)	30 (71%)	12 (29%)
FreeBSD	36 (67%)	18 (33%)	48 (45%)	58 (55%)
JBossas	36 (95%)	2 (5%)	64 (86%)	10 (14%)
PostgreSQL	9 (90%)	1 (10%)	17 (89%)	2 (11%)
Python	20 (91%)	2 (9%)	36 (84%)	7 (16%)
Samba	33 (92%)	3 (8%)	55 (76%)	17 (14%)

7.4.2 RQ_2: To what extent would it be possible to recommend mentors to newcomers joining a software project?

To evaluate the recommendation method, we split the history of each project in a training set and test set. The bottom part of Table 7.1 reports, for each project the time interval of training and test sets, the number of mentors identified in the training set(s), and the number of newcomers in the training set(s) and test set(s). As explained before, since we used in a (i) first case mailing lists data only, and in a second case (ii) both mailing lists and issue trackers data we have two test sets and two training sets as reported in Table 7.1.

Table 7.4 and Table 7.5 report the accuracy of YODA in recommending mentors for the newcomers of the test sets when considering mailing lists data only and both mailing lists and issue trackers respectively. We recommend mentors among those identified—in the training set—using $0.5 \cdot f_1 + 0.25 \cdot f_2 + 0.25 \cdot f_3$, which as explained in $\mathbf{RQ}_1$ exhibits in most cases the best performances. In both the tables the left-side of each column reports the number and percentage of correct and incorrect top 1 mentor recommendations for each newcomer (ranked according to Dice asymmetric similarity), while the right-side reports the number and percentage of correct and incorrect top 2 recommendations.

As the Table 7.4 shows, the percentage of correct top 1 recommendations is very high, above 80% except for Python, where it is 65%, and for FreeBSD, where it is 30% (for the reasons explained in $\mathbf{RQ}_1$). Even when we consider the top 2 recommendations, the correctness remains high, 76% or above—including this time Python where it increases to 77%—and again with the exception of FreeBSD where it is 24%. Consistently with the results achieved in Table 7.4, in the Table 7.5 we find that the percentage of correct top 1 recommendations is higher than 80%, except for FreeBSD, where it is 65% (for the reasons explained in $\mathbf{RQ}_1$). Even when we consider the top 2 recommendations, the correctness remains very high, 71% or above and again with the exception of FreeBSD where it is 45%.

$\mathbf{RQ}_2$ summary:

- ***Mailing lists data**: YODA is able to recommend mentors with a correctness of about 65% or above for the top 1 recommendations and 77% or above for the top 2 recommendations for all systems except FreeBSD, where the percentages decrease to 30% and 24%, respectively.*
- ***Mailing lists and Issue Trackers data**: YODA is able to recommend mentors with a correctness of about 80% or above for the top 1 recommendations and 71% or above for the top 2 recommendations for all systems except FreeBSD, where the percentages decrease to 67% and 45%, respectively. Moreover, the use of two sources of information helps to identify an higher number of mentors in each project.*

.

7.4.3 $\mathbf{RQ}_3$: How does mentoring activity affects the future trajectory of a newcomer when she joins the project?

Table 7.6 reports, for the seven projects we analyzed, the number of newcomers that were supported by mentors (N_m), the number of newcomers that did not receive any support (N_a). Moreover, the table reports, the number of newcomers that had a mentor and became LCT ($N_{m_{LCT}}$); the number of newcomers that had not a mentor and became LCT ($N_{a_{LCT}}$). Then, the table reports the average time of the permanence of each set of LCTs.

The results confirm the general finding of previous work that the majority of developers (except JBossas and PostgreSQL) that join a project usually do not receive support by experienced developers [19, 22]. However, this percentage vary and depends very often from the project size. For example, in PostgreSQL, the smaller project in terms of number of developers, the majority of newcomer have been supported by mentors of the project. Clearly, in

Table 7.6: Joiners and LTCs.

	Newcomers		**LCT**		**Avg. Time (in years)**		
SYSTEM	N_m	N_a	$N_{m_{LCT}}$	$N_{a_{LCT}}$	$N_{m_{LCT}}$	$N_{a_{LCT}}$	***OR***
Apache	20	54	20 (**100%**)	41 (74%)	**11,32**	5,2	**6,87**
Eclips CDT	39	45	35 (**90%**)	29 (64%)	**6,19**	4,31	**4,74**
FreeBSD	80	197	80 (**100%**)	158 (80%)	**9,78**	5,28	**19,63**
JBossas	70	39	65 (**93%**)	31 (79%)	**6,52**	4,55	**3,31**
PostgreSQL	18	5	18 (**100%**)	4 (80%)	**9,78**	5,28	**4,13**
Python	42	54	42(**100%**)	33 (61%)	**9,45**	4,08	**26**
Samba	42	51	33 (**79%**)	20 (39%)	**7,06**	3,04	**5,56**
Average			**95%**	**57%**	**8,59**	**4.53**	**10,03**

this case this results is obvious because when the number of newcomers is very low senior developers have more time to train them. What is more interesting is to observe in Table 7.6 is that the percentage of newcomers that become LCTs is higher for developers that had at least a mentor(s) during the first period in the project. This finding is confirmed by the results of the *OR*: (i) for all the project newcomers that had a mentor has, at least, three time more chances to become LCTs with respect to newcomers that did not received any support; (ii) considering all the projects, in average, this probability is superior, i.e., around 10 times. Moreover, this finding correspond in a higher permanence (longevity in the project) of a newcomer in the project. Specifically, as reported in Table 7.6 the average permanence (in years) of these developers that are mentored by senior developers in the project is almost twice that of developers that do not receive any initial support.

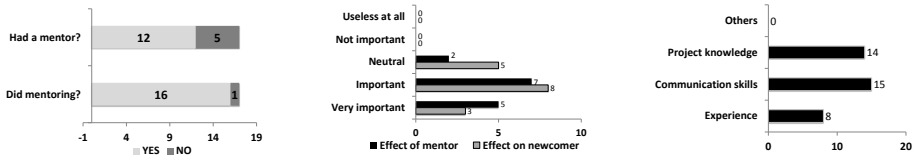
RQ$_3$ summary: *The results of our study, in agreement with others previous work, feed the needed of mentoring of newcomers in software projects. A properly training by project mentors impacts the trajectory (career) of newcomers that are join the project. Well trained newcomers have an higher permanence in the project, almost twice that of developers that do not receive any initial support.*

7.5 Discussion

In the following, we provide additional, qualitative insights to the quantitative study reported in Section 7.4.

7.5.1 Hints collected from project contributors

This section reports results we collected surveying project developers. Since we collected a limited number of answers (17), we report aggregate data only for the purpose of explaining how the respondents perceived the importance of the mentoring process.



(a) Done/received mentoring

(b) Perceived importance of mentoring

(c) What makes a good mentor

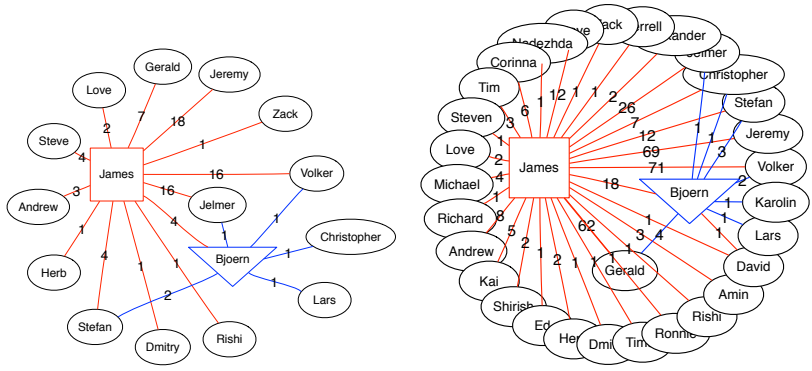
Figure 7.3: Survey questionnaire answers: generic questions on mentoring activity and its importance.

Figure 7.3(a) indicates that, out of 17 developers, 16 admitted to have been mentors, and 12 indicated that they were mentored by someone else. Figure 7.3(b) reports the perceived importance of mentoring when respondents performed mentoring themselves or when they were mentored. The effect was perceived very important (4) or important (9) for developers performing mentoring, while when developers received mentoring, 6 developers indicated that such activity is very important and 5 that it is important. In summary, developers indicated that mentoring is important, although it seems that developers are more likely to admit that they performed mentoring than they were mentored. Also, mentoring seems to be perceived more important by developers that performed it rather than by developers that received it.

Figure 7.3(c) reports what factor developers indicated to be important for good mentoring. The respondents suggested it is mainly matter of communication skills (15) and project knowledge (14), while only 8 developers indicated that experience can play an important role. However, one of the respondents said *"Maybe good communication skills is the least important of the three above, if one is working closely together."*, thus the importance of communication skills depends of the cohesiveness of the project team.

Some developers added comments to the survey to indicate that they performed/received mentoring using communication means different from emails. For example, they added comments like *"Suggested work they can do. But mostly over IRC."*, or *"Not only mailing lists, but also IRC and direct communication."*

By classifying developers in the list we provided, the respondents indicated us a set of 38 mentor-newcomer pairs. We identified 18 of them correctly (the remaining 20 were not identified), while we identified 3 pairs for which respondents indicated that mentoring did not occur. Thus, with respect to the answers provided in the survey questionnaire, our approach has a precision of 18/(18+3)=86% and a recall of 18/38=47%.



(a) One week after Bjoern approached the project

(b) One year after

Figure 7.4: Example (from Samba) of developers' network involving a newcomer (Bjoern) and a mentor (James).

7.5.2 Examples of cases where YODA worked well and where not

In the following we report some examples of collaborations and fragments of emails indicating cases in which YODA worked well to identify mentors (**RQ1**) and cases where it did not. Cases of *true positives*—confirmed by respondents of our survey—were made evident by exchanges of email in which the newcomer clearly asked for help.

Figure 7.4 shows an example of newcomer (Bjoern)-mentor (James) network found in Samba, when the newcomer approached the project by sending his first emails (Figure 7.4(a)), and after one year (Figure 7.4(b)). Note that edges are labeled with the number of exchanged emails. James had already a three-years experience (with 150 commits performed) when Bjoern approached him. As it can be seen from the network, James has a high degree (=13), and Bjoern mainly exchanges emails (4 emails exchanged) with James rather than with other people. After one year, the social importance of James increase, as well as (to some extent) the one of Bjoern, which however still has James as main contact (18 emails exchanged). The mentoring relationship is also evident from fragments of their communication: *"Hi James, maybe you can bring some light into the dark here: I did some tests with . . . "*

Another example (confirmed by the survey respondent) is the one in Samba where Kai asked Jeremy[12] help because she wanted to contribute to the project *"As you might know, I'm a . . . student implementing NTLMSSP signing/sealing in Wine. I've worked on basic NTLM authentication for Wine last year, using ntlm_auth. . . . I decided to give Samba4's GENSEC subsystem a try."*. Then, Jeremy responded with a detailed list of instructions answering Kai

[12]We do not report last names for confidentiality reasons.

specific technical questions, e.g. *"This is the quickest way to make this work (IMHO)."*, *"Probably second best solution."* (where Kai proposed two possible implementations), or even discouraging Kai to implement some features *"this will be a long long road to walk..."*

One example of *false negative* is when the newcomer asked for help and the mentor never (or seldom) responded. An instance of such a case was reported as a true mentoring in our survey, however YODA did not detect it because of the low value of f_1.

One example of *false positive* concerns the collaboration between Ivan and Robert in FreeBSD. Robert helped Ivan to deal with some performance problem of the SSH protocol implementation. Although also from the email it appeared clear that Robert—indicated as candidate mentor by our approach—was the expert giving suggestions, very likely he helped to solve a specific problem only, so to be considered as someone who helped Ivan, while not really Ivan's mentor.

Now we report hereby one case where mentoring recommendation (**RQ2**) worked and one when it did not. When Lucien joined the Apache httpd project, he asked information about proxy handling *"I'm translating mod_proxy.xml, and I don't understand what the term worker means ..."*. Then, after he received information about the meaning of this term, he checked whether he correctly understood: *"So, say we have this proxy configuration: ProxyPass path1 server1, ProxyPass path2 server2 ..."*. In summary, Lucien seems to be interested to work on tasks related to proxy. YODA recommended him Joshua as a mentor, which in the past helped someone else (Laurent) on a problem related to proxy: *"ProxyPass-Reverse only rewrites Location: headers ...."*. That is, the email referred several times to the terms *Proxy* and *Pass*, also contained in the early email sent by Lucien.

An example of *false positive* is between Takashi (newcomer) and Joshua (candidate mentor). Both the early emails of Takashi and previous emails of Joshua contained some common terms, such as "3D", contained in XML and HTML fragments, e.g., *"<note type=3D "warning">"* in the early email by Takashi and *"<meta name=3D "generator"/ >"* in previous emails by Joshua. However, the technical content of the emails was not very similar.

7.6 YODA Limitations and Threats to Validity

This section describes the limitations of YODA and the threats to validity of the study we performed. We are aware that YODA is limited for the following reasons:

- It identifies mentor-newcomer pairs mainly based on their activity on mailing lists and issue trackers. We are aware, however, as also reported in a paper by Aranda and Venolia [34], that developers can communicate also outside mailing lists, e.g., using a chat. One of the developers that responded to our survey *"I've asked questions personally*

in private chats." and when we asked if he helped someone else, he answered *"Yes, suggested work they can do. But mostly over IRC."*.

- It does not account for the availability of mentors. However, since YODA proposes a list of mentors for a newcomer, the project manager can easily assign to the newcomer a mentor that is currently available.

- It matches the expertise required by the newcomer with the expertise of possible mentors by comparing the early emails (or issue trackers messages) of the newcomer with all emails (and all issue trackers messages) of the mentor. However, as we explained in Section 7.2.3, it is not the purpose of this paper to propose a better method to identify expertise, as others have been proposed already in the past [189, 191, 192].

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. First, there could be imprecisions in the way we mapped mailing list names and issue trackers with versioning system IDs. However, we manually validated this mapping. In addition, for systems using git (Apache httpd, Eclipse CDT, PostgreSQL, Python, and Samba) the mapping is straightforward in that full names and emails are used as git IDs.

The "age" of a developer is computed by observing the first email (or first trackers messages) exchanged, which might have occurred way before (or way after) the person joined the project. Another aspect of our validation that is inherently subject to imprecision is the manual validation of YODA recommendations. We limited the degree of error-proneness and subjectiveness by having two different people performing the inspection independently.

Threats to *internal validity* concern external factors, we did not consider, that could affect the variables being investigated. The factors we considered, f_1–f_5, are only a partial view of the mentor and newcomer experiences, and of their inter-communication. There can be other factors we did not consider. Future work will be devoted to consider other factors aiming at mitigating such a threat.

Threats to *external validity* concern the generalization of our findings. We have performed our study on data from seven different systems belonging to different domains and having different size in terms of code base and number of developers. Further evaluation is however necessary, especially in industrial environments where there can be a tacit knowledge—within an organization, but not encoded in the communications—of who can be a good mentor and who not.

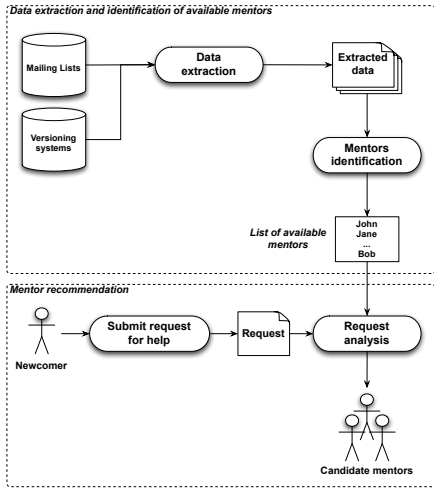


Figure 7.5: YODA in Eclipse: information flows.

7.7 YODA tool support

This Section describe YODA (Young and newcOmer Developer Assistant), an Eclipse plugin that implements the approach described in Section 7.2 and that, based on the project source code repository, its mailing list, and a query the newcomer explicitly makes (e.g., "I need help on the network component protocol"), or a query inferred from the files opened by the newcomer, is able to recommend appropriate mentors.

Section organization: subsection 7.7.1 introduces the Yoda Eclipse plugin and shows it in action on a concrete scenario taken from the Samba project.

7.7.1 Integrating YODA in Eclipse

Fig. 7.5 depicts the YODA Eclipse plugin flow of information. Specifically, YODA extracts communication information by parsing mailing lists downloaded from the project mailing list archives, and retrieves information about changes performed by developers by retrieving the commit logs from the versioning systems. Such information is stored in a fact database, and is used to identify a list of candidate mentors to be recommended to newcomers/project managers.

To use YODA, the developer (or project manager) has to set some preferences (including various YODA parameters). Then, as shown in Fig. 7.6, she can mine data from mailing lists and from the versioning system to identify mentoring relationships in the past project history.

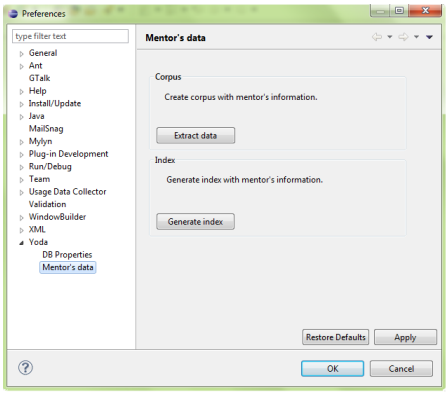
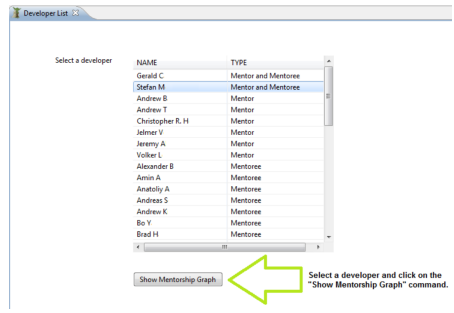


Figure 7.6: Mining candidate mentors from software repositories.

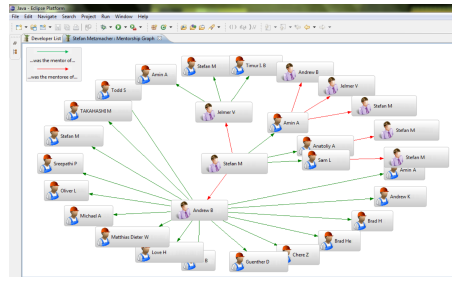
In the following, we will describe how YODA can be used from two perspectives: (i) of a *project manager*, who wants to help various newcomers in the project, recommending them appropriate mentors, and, in general, is interested to monitor project collaborations to understand who is assuming a leadership role; and (ii) of a *project newcomer*, who is willing to contribute to a feature involving some specific source code files, and needs some support for her work. Our demonstration is based on data from the Samba project. For privacy reasons we have anonymized last names in the screenshots.

Concerning the first perspective, the project manager—by selecting the "Mentorship graph" from the YODA menu—can access a tab with a list of developers (Fig. 7.7(a)), and select a developer for which one wants to understand the role played in the project. The list shown in this tab already indicates whether the developer played the role of *mentor*, *mentoree*, or both. After selecting a developer, it is possible to visualize a collaboration graph (Fig. 7.7(b)), that starts from the selected developer and represents mentor/mentoree relationships up to a given distance set in the YODA preference, e.g., only direct relations if the distance is one, mentor/mentoree of developers mentored by the selected developers if the distance is two, etc. For a given developer, green arrows are directed towards *mentorees*, while red arrows are directed towards *mentors*. In our example of Fig. 7.7(b), we can see a graph starting from *Stefan M.*, and representing relations towards his mentors (*Andrew B.* and *Jelmer V.*) and towards his mentorees (*Sam L.* and *Anatoliy A.*). It can be noticed that, for the sake of simplicity, the oposite relations (e.g., red arrow between *Sam L.* and *Stefan M.*) are not shown.

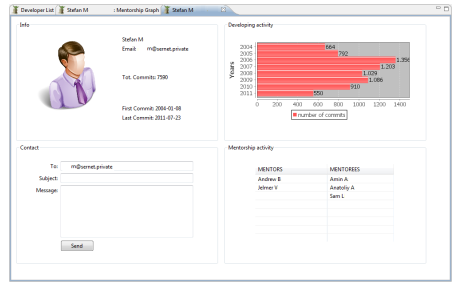
Clicking on a specific developer—say *Stefan M.* in our case—a new tab (Fig. 7.7(c)), is shown, from which it is possible to analyze detailed *social* and *technical* information of the developer. Specifically, it is possible to know when the developer entered the project, the



(a) Selecting a developer



(b) Mentorship graph

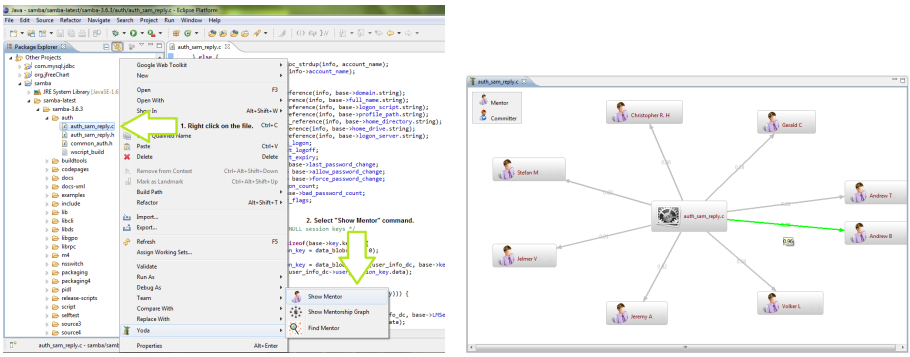


(c) Detailed information about a specific developer

Figure 7.7: How YODA (a) shows mentorship relations in a project and (b) allows to browse information about a developer and to get in touch with him.

number of commits performed over different years, and the list of her mentors/mentorees. If needed, the tool also allows to send an email to this developer.

Let us now see how YODA can be used from a newcomer's perspective. In such a perspective, YODA allows a developer to formulate a request for help in two ways, namely, (i) *implicit query*, based on the context—i.e., source code files—which the newcomer is (in-



(a) Seeking experts about specific source code files (b) Getting available mentors with the desired expertise

Figure 7.8: Mentor recommendation using an implicit query based on the context which the developer is working on.

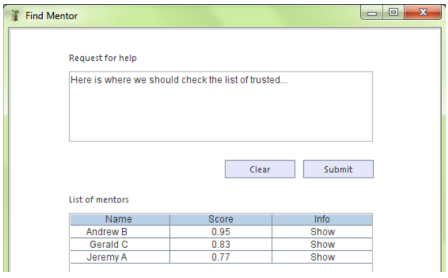


Figure 7.9: Explicit (natural language) request for mentor.

terested to) working on; or (ii) *explicit query*, i.e., by writing a natural language sentence expressing the need for help on a particular topic, component, etc.

As for the implicit query, let us suppose a developer is working on a specific source code file, say *auth_sam_reply.c*. By right-clicking (Fig. 7.8(a)) on the file name, and by selecting the "Show mentor" from the YODA menu, it is possible to identify developers that have enough expertise for such a file. The edge between the file and the developer is labeled with a confidence value (between 0 and 1) corresponding to the Dice similarity between the (implicit) query and the developer corpus. Among the various developers, YODA can recommend—with a different, green edge, and with a different icon—the availability of people that, besides being expert on the file also demonstrated to be good mentors in the past.

As for the explicit query, let us suppose a developer has a specific request for help. By selecting the "Find mentor" from the YODA menu, it is possible to submit a request for help and identify developers that have enough expertise for such a request (Fig. 7.9). Specifically,

once submitted the request for help, YODA provides the newcomer with the list of candidate mentors ranked according to the Dice similarity between the query and the developer corpus.

7.8 Related Work

There have been several researchers investigating what happens when a newcomer joins a software project, and which are the factors for her growth, including mentoring. Our work falls in the general area of those aimed at relating social relations among developers with technical aspects of the project they are working on.

Dagenais *et al.* [25] studied, by surveying 18 IBM developers, what happens when someone moves into a new "project landscape", making for her necessary to get acquitted with the new environment. Among the factors they found important, it is worthwhile to mention the need for early practicing, the availability of feedback for their work, and *the need for getting initial guidance*. The latter is totally in agreement with what we collected from our small survey, and motivates approaches like YODA. Fronza *et al.* [196] studied how newcomers join agile projects, finding that pair-programming is used to initiate newcomers to a project.

Zhou and Mockus [22] investigated, on three industrial and three open source projects, how the sociality level of a project in a particular moment influences the likelihood for newcomers to become long-term contributors. They suggest the need for proper training plans for newcomers in open source projects: this requires to identify appropriate mentors. Zhou and Mockus [197] also identified challenges in a software market where offshoring, outsourcing and open source development are increasing fast: understanding cultural differences, analyzing how developers grow their expertise, and *providing tools to facilitate newcomers' learning*, as YODA does.

A different perspective of developers' growth in software projects was studied by Sinha *et al.* [198] who investigated, on the Eclipse project, how project contributors become committers, and found that this depends on having contributed patches/source code of the project, being active in other open source projects, or being part of the project organization. Also, Bird *et al.* [199] investigated the phenomenon of "immigration" in software projects, explaining the *who, how and when* in the process of providing to newcomers the authority to commit in the project repository.

In summary, while all these works highlighted the need for supporting newcomers when they join a software project, to the best of our knowledge this is the first work aimed at proposing an approach to identify mentors in software projects and to recommend them to newcomers.

In the past and recent years, other kinds of recommenders have been proposed to support

developers—especially junior ones. Among others, Hipikat [200] provides recommendations about components relevant for the current coding context of the developer. As discussed in Section 7.2.3, identifying mentors also requires the selection of people having specific expertise. This has been done by Anvik *et al.* [189], by Canfora and Cerulo [191], and by Tamrawi *et al.* [192] who proposed approaches—based on machine learning (IR), IR, and fuzzy clustering respectively—for bug triaging, i.e., to determine the most suited developers able to fix an incoming bug.

7.9 Summary

Mentoring is particularly important to make software project newcomers knowledgeable of various aspects of the project on which they are contributing, such as technical details, coding guidelines, and organizational rules. As part of our study we contacted contributors to open source projects, one of which mentioned *"My general view is that it is very important that mentor and mentee share at least the mindset and the same passions. My primary mentors have been . . . and they both had a very strong technical background, something I definitively wanted to match."* This Chapter proposed YODA, an approach to identify mentors by relying on historical data from a software project, and then recommend them when a newcomer joins the project. Being inspired by *ArnetMiner*—a tool that analyzes academic collaborations—YODA identifies mentoring when the newcomer exchanges most of her emails with the mentor, and the mentor has a higher social importance and project age than the mentor. Then, when a newcomer joins a project, her early discussion is used to identify the expertise she is requiring, and YODA recommends, among the candidate mentors previously identified, those who exhibited a discussion (textually) similar to the newcomer early discussion. YODA has been evaluated on data from seven open source projects, Apache HTTPD, Eclipse CDT, FreeBSD, JBoss AS, PostgreSQL, Python, and Samba. Results of the study indicate that, except for FreeBSD, YODA is able to identify candidate pairs of mentor-newcomer with a precision in most cases higher than 80%, and recommend them with a precision greater than 70%. Comments collected from project developers indicated us that mentoring is important and depends on project knowledge and communication skills more than on experience. Finally, results of our study, in agreement with other previous work, feed the needed of mentoring of newcomers in software projects. A properly training by project mentors impacts the trajectory (career) of newcomers that are join the project. Newcomers trained by mentors of the project have an higher permanence (in year) in the project, almost twice of developers that do not receive any initial support.

Chapter 8

Mining Source Code Descriptions from Developer Communications

Contents

In the Chapter 7 we propose an approach, named YODA (**Y**oung and newc**O**mer **D**eveloper **A**ssistant) aimed at identifying and recommending mentors in software projects by mining data from mailing lists and versioning systems. Candidate mentors are identified among experienced developers who actively interact with newcomers. Then, when a newcomer joins the project, YODA recommends her a mentor that, among the available ones, has already discussed topics relevant for the newcomer. Results indicate that top committers are not always the most appropriate mentors, and show the potential usefulness of YODA as a recommendation system to aid project managers in supporting newcomers joining a software project.

However, other kind of recommenders can be used to support developers in other task, like for example, in code re-documentation. As discussed in the Chapter 6 source code summaries can be a valid source to understand source code. However, very often, source code lacks comments that adequately describe its behavior. In such situations developers need to infer knowledge from the source code itself or to search for source code descriptions in external artifacts.

We argue that messages exchanged among contributors/developers, in the form of bug reports and emails, are a useful source of information to help understanding source code. However, such communications are unstructured and usually not explicitly meant to describe specific parts of the source code. Developers searching for code descriptions within communications face the challenge of filtering large amount of data to extract what pieces of information are important to them. We propose an approach to automatically extract method descriptions from communications in bug tracking systems and mailing lists.

This Chapter presents a recommender useful for code re-documentation. The usefulness of such a tool is motivated by the fact that, very often, source code lacks comments that adequately describe its behavior. In practise, such approach mine messages exchanged among contributors/developers, in the form of bug reports and emails, and extract useful descriptions, that describe specific parts of the source code. We have evaluated the approach on bug reports and mailing lists from two open source systems (Lucene and Eclipse). The results indicate that mailing lists and bug reports contain relevant descriptions of about 36% of the methods from Lucene and 7% from Eclipse, and that the proposed approach is able to extract such descriptions with a precision of up to 79% for Eclipse and 87% for Lucene. The extracted method descriptions can help developers in understanding the code and could also be used as a starting point for source code re-documentation.

8.1 Motivation: incomplete and unclear code comments need to be re-documented

Consider the following situation. A newcomer is reading the Java code of an unfamiliar (part of the) system. She encounters a methods call. Ideally, a good method name would indicate its purpose. If not, a nice Javadoc comment would explain what the goal of the method is. Unfortunately, the method name is poorly chosen and there are no comments. Not an uncommon situation. At this point, the developer has the choice of reading the implementation of the method or searching the external documentation. It is very rare that external documentation is written at method level granularity (especially when comments are missing) and that such specific information is easy to retrieve. The goal of our work is to help newcomers in such situations. Specifically, we aim at providing newcomers with a means to quickly access descriptions of methods.

Our conjecture is that, if other developers had any issues related to a specific method, then a discussion occurred and someone described the method in the context of those issues. For example, developers and project contributors communicate with each other, through mailing lists and bug tracking systems. They often "instruct" each other about the behavior of a method. This can happen in at least two scenarios. First, when a person (sometimes a newcomer in the project) is trying to solve a problem or implement a new feature, she does not have enough knowledge about the system, and asks for help. Second, when a person explains to others the possible cause of a failure, illustrating the intended (and possibly also the unexpected) behavior of a method. For example, we report a paragraph for issue #1693 posted on the Lucene Jira bug-tracking system[1]:

> "new method added to AttributeSource: addAttributeImpl(AttributeImpl). Using reflection it walks up in the class hierarchy of the passed in object and finds all interfaces that the class or superclasses implement and that extend the Attribute interface. It then adds the interface- instance mappings to the attribute map for each of the found interfaces."

which clearly describes the behavior of the *AttributeSource: addAttributeImpl(AttributeImpl)* method.

We claim that *unstructured communication between developers can be a precious source of information to help understanding source code.*

So, why newcomers could not use simple text search techniques, based on text/regular expression matching utilities, such as, *grep*, to find method descriptions in communication

[1] https://issues.apache.org/jira/browse/LUCENE-1693

data? Such simple text matching approaches could only identify sentences having a method name, or in general any regular expression containing the method name plus other strings such as the class name or some parameter names. They would generate too many false positives. As it happens for requirement-to-code traceability recovery [170, 201], such a simple matching is not enough.

This study presents and validates an approach to automatically mine source code descriptions (in particular method descriptions) from developer communications, such as, emails and bug reports[2]. It also presents evidence to support our assumption that developer communications are rich in useful code descriptions.

Our approach traces emails to classes, identifies affirmative textual paragraphs in these emails, and traces such paragraphs to specific methods of the classes. Then, it uses heuristics (based on textual similarity between paragraphs and methods, and on matching method parameters and other keywords to paragraphs) to filter out candidate method descriptions.

The filtering technique results in a set of one or more paragraphs describing each method (for which a description was found). These paragraphs may overlap, in terms of content, or they could describe different aspects of the method behavior, e.g., one describes the method interface and return value, another the behavior in terms of calls to other methods, another the exceptional behavior, etc. The technique retrieves all these paragraphs and combines them into a method description. Such descriptions can have multiple uses:

[1] They can be used as such to help newcomers understanding the code.

[2] In perspective, an automatic tool can further process the descriptions and automatically generate method documentation, e.g., API descriptions or comments.

We have applied the proposed approach to 26,796 bug reports from the Eclipse project, and 18,046 emails and 3,690 bug reports from the Apache Lucene project. Results indicate that emails and bug reports contain descriptions for about 7% of the Eclipse methods and 36% of the Apache Lucene methods. The proposed filtering approach is able to correctly identify method descriptions in 79% of the cases for Eclipse and 87% of the cases for Lucene. Finally, we report several examples describing how methods are likely described in the developers' communication, discussing the linguistic patterns we found in Eclipse and Lucene for different kinds of method descriptions.

However, very often useful methods description are available also in the so called question−and−sites such as StackOverflow. For this reason we adapted our approach in the context of this

[2]For simplicity, we will only refer to mailing lists/emails, although the approach is applicable to bug tracking systems and other similar communications. Only where it matters we will refer to mailing lists and bug tracking systems separately.

question and answer site to verify whether the proposed approach that is originally conceived to mine method descriptions from mailing lists and issue tracker discussions is able to extract useful descriptions also in StackOverflow. Thus, we implemented CODES (mining sourCe cOde Descriptions from developErs diScussions), an Eclipse plugin that automatically extracts Java method descriptions from SO discussions.

The study is organized as follows. Section 8.2 describes the proposed approach. Section 8.3 reports the empirical evaluation using data from Eclipse and Apache Lucene, while Section 8.4 discusses some example of method descriptions found by the approach. Section 8.5 describes CODES, while Section 8.6 discusses the related work. Finally, Section 8.7 summaries the results of our empirical study.

8.2 Mining Method Descriptions from Communications

This section describes the proposed approach for mining method descriptions in mailing lists or bug reports.

8.2.1 Step 1: Downloading emails and tracing them onto classes

First, we download mailing list archives and all bug reports concerning the analyzed time period of the investigated projects. Then, we extract the body from emails using a Perl mailbox parser (*Mail::MboxParser*). Bug reports (downloaded in HTML) are first rendered using a textual browser *lynx* and then the text is extracted using a Perl script. Then, we trace emails onto source code classes (referring to a system release before the email date). For this purpose, we use two heuristics:

[1] We use an approach similar to the one proposed by Bacchelli *et al.* [202, 203]. More specifically, we assume there is an explicit traceability link between a class and an email whenever (i) the email contains a fully-qualified class name (e.g., *org.apache.lucene.analysis.M*
or (ii) the email contains a file name (e.g., *MappingCharFilter.java*)—provided that there are no other files with the same name, or that the file name is also qualified with its path.

[2] For bug reports, we complement the above heuristic by matching the bug ID of each closed bug to the commit notes, therefore tracing the bug report to the files changed in that commit (if any are found).

8.2.2 Step 2: Extracting paragraphs

During a preliminary investigation we determined—by inspecting emails from out case studies—that paragraphs describing different aspects of the email topic are separated by one or more white lines. Therefore, we use such heuristics to split each email into paragraphs. For bug reports, different posts related to the same bug report are treated as separated paragraphs.

Emails often contain source code fragments and/or stack traces that should be pruned as we are interested to mine descriptive text only (in future, we plan to keep such code fragments into account to better link paragraphs to source code). To remove them, we used an approach inspired to the work of Bacchelli *et al.* [204]. We computed, for each paragraph, the number and percentage of programming language keywords and operators/special characters (e.g., curly braces, dots, arithmetic and Boolean operators, etc.). Paragraphs containing a percentage of keywords and special characters/operators higher than a given threshold are discarded. Calibrating such a threshold requires a trade-off. We adopt a conservative approach and are willing to accept losing a few good paragraphs. Similarly to what also shown in the study by Bacchelli *et al.* [204], we found a threshold of 10% to be the best compromise between losing good paragraphs and keeping source code fragments. Remember that our goal is to provide precise descriptions to the developer in order to save time and effort.

A further processing—performed using the English Stanford Parser[3] [205]—aims at preserving only paragraphs in the affirmative form, removing those in interrogative forms—because we assume that method description should not contain interrogative sentences—as well as pruning out sequences of words that the parser was not able to analyze, i.e., sequences of words that cannot be considered valid English sentences.

8.2.3 Step 3: Tracing paragraphs onto methods

To trace paragraphs onto methods, we first extract signatures for all methods in a system version released before the email being analyzed. This is done using the Java *reflection* API.

Then, we identify the paragraphs referring a method. These paragraphs shall meet the following two conditions:

- They must contain the keyword “method”. This is because we are searching sentences like “The method foo() performs...”. Indeed, there could be cases where the method is referred and described in a sentence not containing the keyword “method” (e.g., “foo() performs...”). However, we observed in a preliminary analysis that such cases occur mostly when the method is mentioned in other contexts (e.g., describing a fault) rather than when communicating a method description to other people.

[3] http://www-nlp.stanford.edu

- They must contain a method name, among the methods of classes traced to the email in *Step 1*. We also require that such a name must be followed by a open parenthesis—i.e., we match "foo(" while we do not consider "foo". This is to avoid cases when a word matches a method name, while it is not intended to refer to the method. For example, we found several paragraphs like that e.g., "Method patch", where "patch()" was actually a method of a class traced onto the email.

It is important to note that such a process can be subject to ambiguities. First, an email can be traced onto multiple classes, having one or more method with the same name (and maybe even the same signature). In such a cases, the paragraph is assigned to all of these classes. Second, there may be overloaded methods. Where possible, this is resolved by comparing the list of parameter names mentioned in the paragraph with the list of parameters in the method signature as extracted from the source code. When this is not sufficient to resolve the ambiguity, we conservatively assign the paragraph to all matched methods. As explained later (*Step 5*) both ambiguities can be mitigated by computing the textual similarity between the paragraph and the method.

8.2.4 Step 4: Filtering the paragraphs

We defined—based on the manual inspection of hundreds of emails— a set of heuristics to further filter the paragraphs associated with methods.

These heuristics encode some observed rules of discourse commonly used by developers in emails. The first heuristic concerns *method parameters*: it is required that, if a method has parameters, at least some of them are mentioned in the method description. We count the number and percentage of method parameter names mentioned in the paragraph. We define a score, s_1 as the ratio between the number of parameters mentioned and the total number of parameters in the method. We consider $s_1 = 1$ if the method does not have parameters.

We defined three additional heuristics that captures characteristics of three different categories of method descriptions, i.e., syntactic descriptions, description of how a method overloads/overrides another one, and descriptions of how a method performs its task by invoking other methods.

[1] *Syntactic descriptions (mentioning return values)*: if a method is not *void*, we check whether the paragraph contains the keyword "return". We define a score s_2 equal to one if the method is *void*, or if is not *void* and the paragraph contains "return", zero otherwise.

[2] *Overriding/Overloading*: keywords such as "overload" or "override" are likely to be contained in some paragraphs describing methods. This, in particular, happens when a

paragraph describes the additional behavior with respect to the overridden/overloaded method. We define a score s_3 equal to one if any of the "overload" or "override" keywords appears in the paragraph, zero otherwise.

[3] *Method invocations*: when a paragraph describes a method, often it describes it in terms of invocation of other methods. Therefore, we mine the paragraphs containing for the words "call", "execute", "invoke". We define a score s_4 equal to one if any of the "call", "execute", or "invoke" keywords appears in the paragraph at least once, zero otherwise.

We apply the above described heuristics by constraining the set of selected paragraphs such that $s_1 \geq th_P$ and $s_2 + s_3 + s_4 \geq th_H$, where th_P is a threshold we set for the parameter heuristic and th_H is a threshold for the other heuristics. Details about the two thresholds are reported in Section 8.3.1.

8.2.5 Step 5: Computing textual similarities between paragraphs and methods

After having filtered paragraphs using the heuristics, we rank them based on their *textual* similarity with the method they are likely describing. The rationale is that, other than the method name, parameter names, and other keywords identified in *Step 4*, such paragraphs would likely contain other words (e.g., names of invoked methods, variable names, local variables, etc.) contained in the method body. Also, as mentioned above, computing such a similarity would help mitigating ambiguities when tracing paragraphs onto methods.

To this aim, we extract the method's text from the system source code (again, referring to a version before the email). This is done using the *srcml* analyzer [206]. Then, we normalize the method text removing special characters, English stop words, and programming language keywords, and splitting the remaining words using the camel case heuristics. A similar text pruning is performed on paragraphs. After that, we index the paragraphs and the methods using a Vector Space Model implemented with the R[4] *lsa* package.

We compute the textual similarity between each paragraph P_k and the text of each traced method M_i using the cosine similarity [80]. For each method M_i, we rank its relevant paragraphs P_k by the similarity measure. Finally, we consider only the paragraphs that have a similarity measure higher than a threshold th_T. These are the paragraphs that are presented to the user as containing a description to the method M_i. As it will be shown in Section 2.3, varying th_T will produce different results in terms of precisions and of retrieved candidate method descriptions.

[4] http://www.r-project.org

8.2.6 Limitations of the proposed approach

Our proposed approach—to the best of our knowledge—represents the first attempt to mine method descriptions from developers' communication. As with any work that addresses a problem in premiere, limitations exist, which we hope to address in future work. We highlight here those that should be kept in mind while interpreting the results of our empirical evaluation from the next section:

- *We do not consider sequences of paragraphs that can describe the same method.* In some cases, a method description can be longer than one paragraph and thus span over multiple paragraphs. A preliminary attempt at clustering subsequent paragraphs to the ones that mention a method drastically lowered the precision of our approach. More sophisticated approaches are needed to address this issue.

- *Paragraphs often describe partial/exceptional behavior.* In some cases, the paragraphs describe only part of the method behavior, because the communication concerns only that part. In other cases, the paragraphs describe exceptional behavior. We believe that such paragraphs are useful to the developers to get a complete or partial overview of a method's syntax and behavior.

- *We do not really mine abstractive method description, but rather extractive descriptions.* Since our technique uses textual similarities, it will recover paragraphs describing a method behavior only if this is done using (some of) the elements (e.g., names of invoked methods, method parameters, local variables, etc.) contained in the method body—a.k.a, extractive description. For example, our approach may not recover a paragraph providing a high-level description of an algorithm (e.g., imagine a paragraph describing a sorting algorithm), which uses terminology not used in the implementation of the method—a.k.a., abstractive description. However, mixed descriptions will likely be retrieved by the approach.

8.3 Empirical Evaluation

The *goal* of this study is to evaluate the proposed approach for extracting method descriptions from developers' communications. The *quality focus* is the ability of the proposed approach to cover methods from the analyzed systems, as well as the precision of the proposed approach. The *perspective* is of researchers who want to evaluate to what extent mining developers' communications can be used to support code understanding and to what extent the proposed approach is able to identify method summaries with a reasonable precision.

Table 8.1: Characteristics of the two subject systems.

Characteristic	Eclipse	Lucene
Analyzed Period	2001–2010	2001–2011
KLOC range	283-2,486	6–345
#of classes (range)	4,829–18,834	427–528
#of methods (range)	31,132–117,654	2,432–2,952
# of bug reports	26,796	3,690
# of emails	–	18,045
# of paragraphs from bug reports	202,539	115,504
# of paragraphs from emails	–	91,408
Total # of paragraphs	202,539	206,912

The *context* consists of bug reports from the Eclipse project and both mailing lists and bug reports from the Lucene project. Eclipse[5] is an open-source integrated development environment, written in Java. Lucene[6] is a text retrieval library developed in Java. Table 8.1 reports some relevant characteristics of the two systems and the data we used. While Eclipse can be considered as a large system, Lucene is a small-medium system.

The empirical study reported in this section aims at addressing the following research questions:

- **RQ**$_1$ *How many methods from the analyzed software systems are described by the paragraphs identified by the proposed approach?* While we do no expect to find descriptions for all, or nearly all of the methods, we believe that the approach would be useful in the practice only if finding descriptions for a given method would not be an extremely rare event.

- **RQ**$_2$ *How precise is the proposed approach in identifying method descriptions?* This research question aims at determine whether the mined description are meaningful method descriptions, or whether they are, instead, *false positives*. While some false positives are unavoidable, too many of them would make the approach unpractical.

- **RQ**$_3$ *How many potentially good method descriptions are missed by the approach?* This research question aims at providing an idea of how the proposed approach is affected by *false negatives*, i.e., filtering out good method descriptions.

8.3.1 Threshold calibration

Step 4 of the proposed approach relies on two thresholds, th_P and th_H. To calibrate th_P, we analyze the distribution parameters referred to in the paragraphs traced onto methods.

[5]http://www.eclipse.org
[6]http://lucene.apache.org

For Eclipse, the percentage of parameters had a minimum and first quartile equal to zero, a median=50%, and a third quartile and maximum equal to 100%. We selected the median as threshold and analyzed the performance with different settings for th_P varying it between 0% and 100% in 10% steps. We confirmed that the median choice works equally well for Lucene. We realize that such a rule for selecting th_P cannot be easily generalized, but it worked for these two systems and for proof of concept purpose. Investigating the generality of this rule is subject of future work.

Regarding th_H, we set it to 1—i.e., accepting all cases where $s_2 + s_3 + s_4 \geq 1$—in order to select paragraphs containing at least one keyword able to characterize the paragraph with respect to the different kinds of method descriptions outlined in *Step 4* of the approach. Once again, identifying alternative rules for calibration, which generalize better, is subject of future work.

The effect of the third threshold, th_T, on the precision of the approach is analyzed in detail in the following subsection.

8.3.2 Evaluation procedure

First, we extracted, using *Steps 1-3* of the proposed approach, a set of candidate paragraphs that are traced onto methods. We refer to them as the subset of *traced paragraphs*. After that, we performed a first pruning using the heuristics from *Step 4*, i.e., selecting all paragraphs—referred to as *candidate descriptions* from here on—having $th_P \geq 0.5$ and $th_H \geq 1$.

Subsequently, we computed the cosine similarities, with the aim of investigating how the performance of the approach varies by considering only paragraphs having a cosine similarity greater than a given threshold.

Then, we built the oracle against which to validate our results. The oracle was done by manually validating a sample of the *candidate descriptions*. Since it was not possible to manually validate all descriptions, we sampled 250 descriptions for each project. Such a sample allows to achieve estimations with a confidence interval of $\pm 5\%$ assuming a significance level of 95% [207]. We decided not to perform a random sampling of the descriptions: since our aim is to analyze how the precision and the method coverage vary with different thresholds of the cosine similarity, we wanted to include in the sample enough data points representative of different cosine ranges. Therefore, the most appropriate way to proceed was to apply a stratified sampling. We divided our population of candidate descriptions in sets according to five classes of cosine range: 0%-20%, 20%-40%, 40%-60%, 60%-80%, and 80%-100%. Then, based on the distribution of descriptions over the different classes, we randomly sampled 25, 50, 100, 50, 25 descriptions for the five classes, respectively.

Then we asked three reviewers to analyze the sampled descriptions and decide whether

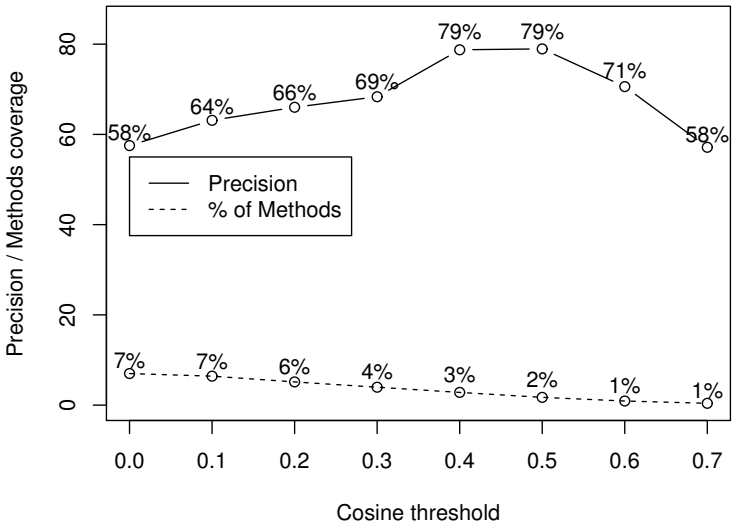
Table 8.2: Number of paragraphs and method coverage after applying filtering from Steps 2, 3 and 4 of the approach.

Filtering	Eclipse		Lucene	
	# of paragraphs	method coverage	# of paragraphs	method coverage
Step 2	202,539	–	206,912	–
Step 3	42,095	22%	12,417	65%
Step 4	3,111	7%	3,707	36%

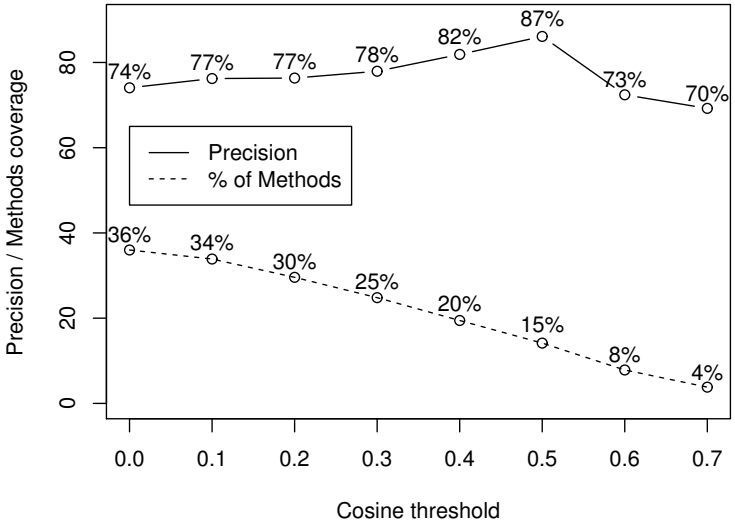
they were, or not, reasonable paragraph descriptions. Two reviewers were two of the work authors (one of which did not know the detail of the mining algorithm at the time the validation was performed, so not to bias such a validation), and the third reviewer was a PhD student not involved in the work. To rate a description, the three reviewers had the system source code available and checked whether the description is, indeed, one possible way a method could be described, either in terms of its syntax, as extension of other methods, or in terms of a method invocation chain. If all three reviewers agreed that a paragraph is a specific kind of description for a method, then the paragraph was classified as true positive. If all three reviewers agreed that a paragraph is not a good description for a method, then the paragraph was classified as false negative. When they disagreed, they discussed until they reached consensus. In the end, **500** paragraphs were included in the oracle.

To address $\mathbf{RQ}_1$ we considered all the methods in the analyzed systems, whereas for $\mathbf{RQ}_2$ we only used the paragraphs in the oracle, in order to analyze how the *method coverage* ($\mathbf{RQ}_1$) and the *precision* ($\mathbf{RQ}_2$) change when increasing the cosine threshold. We define the *method coverage* for a given cosine threshold th_T as the percentage of the methods in the system for which there exists at least one candidate description traced onto it and such that $cos(m_i, P_j) > th_T$. We define the *precision* for a given cosine threshold th_T as the percentage of true positives in the oracle for which $cos(m_i, P_j) > th_T$.

Addressing $\mathbf{RQ}_3$ is more difficult. We are aware that precisely assessing false negatives would be impossible (it would require analyzing the entire body of emails). Instead, we extracted a small sample (100 paragraphs for each project, thus in total further 200 paragraphs) from the set of paragraphs pruned after applying the *Step 3* heuristics, i.e., all paragraphs that can be mapped onto a method (and not all possible paragraphs, because we assume that a paragraph describing a method at least mentions it), but do not satisfy our similarity-based filtering. We manually validated the sample similarly to how we did it for the oracle, in order to compute the percentage of *false negatives* in the sample.



(a) Eclipse



(b) Lucene

Figure 8.1: Precision and method coverage for different levels of similarity. Note that the maximum possible coverage would be (see Step 3 of Table II) 22% for Eclipse and 65% for Lucene.

8.3.3 Results

Table 8.2 reports results about the number of paragraphs obtained after applying steps 2, 3, and 4 of the approach, i.e. (2) the initial set of paragraphs extracted from the emails after pruning out source code and irrelevant sentences (short and interrogative ones), (3) the number of paragraphs traced to methods, and (4) the number of paragraphs traced to methods that satisfy the filtering according to *Step 4* heuristics i.e., paragraphs with $th_P \geq 0.5$ and $th_H \geq 1$. The table also reports, for the last two cases, the percentage of covered methods. As it can be noticed, about 20% of the Eclipse paragraphs and 5% of the Lucene paragraphs can be traced to methods (*Step 3*), which ensures a coverage of 22% of the Eclipse methods and 65% of the Lucene methods. However, such paragraphs do not satisfy the heuristics of *Step 4*, nor they are constrained by any textual similarity threshold.

When applying the heuristics of *Step 4*, the number of paragraphs is reduced to 3,111 for Eclipse and 3,707 for Lucene, which results in 7% method coverage for Eclipse and 36% for Lucene.

Figure 8.1 reports the achieved precision and the method coverage for both Eclipse and Lucene. The x-axis shows the increasing cosine similarity threshold, th_T, while the y-axis shows both the precision and the method coverage. Note that the precision is on a 0-100% scale, whereas maximum coverage for Eclipse is 22% and for Lucene is 65%.

The results for both systems correspond to our expectations, i.e., increasing the cosine threshold results in an increase in precision and it comes at the cost of reduced method coverage. An interesting phenomenon is that the increase in precision peaks (79% for Eclipse and 87% for Lucene) at a threshold of approximately 0.5 for both systems, which means that maximum precision can be achieved without complete loss of method coverage. In both cases, the difference between the minimum and maximum precision is higher than the difference between the minimum and maximum method coverage (proportionally). In other words, the precision gain increases slower than the loss in method coverage. Method coverage in Eclipse for highest precision is about 3% (which means covering between 933 and 3,530 methods, depending on the version), and for Lucene is about 15% (which means covering between 365 and 442 methods).

We can summarize the results related to $\mathbf{RQ}_1$ (method coverage) and $\mathbf{RQ}_2$ (precision) stating that, on the one hand, the proposed approach is precise enough to mine method descriptions, thus reducing the developers' burden to go through a wide number of false positives. On the other hand, the percentage of covered methods could appear as relatively low, thus it is useful to pursue a compromise between coverage and precision. However, it is important to note that (i) we cannot really expect to find descriptions for all methods, especially for large systems like Eclipse (for which, by the way, we do not have emails, but bug reports

only), and (ii) the coverage depends a lot on the quality—with respect to our goal of mining descriptions—of the project discussion, which in our case seems to be better for Lucene than for Eclipse.

Regarding $\mathbf{RQ}_3$—i.e., the presence of *false negatives*—the analysis of a sample of 100 paragraphs traced to methods, but not satisfying the *Step 4* heuristic, indicates that, for Eclipse, 78 out 100 paragraphs have been classified as *true negatives*, leaving 22 paragraphs that could represent good method descriptions, but that were discarded by our heuristics. For Lucene, 67 paragraphs were classified as true negatives, leaving a relatively large (33%) number of *false negatives*. Although this can be seen as a limitation of the proposed approach for capturing good method description, this can be explained by the peculiar characteristics of the Lucene mailing lists and bug reports, which contain many very good method descriptions, as it has also been noticed from the high precision obtained for $\mathbf{RQ}_2$. As stated before, heuristics that result in a better balance between a false negatives and false positive will be investigated in the future. The current results are encouraging enough to motivate future research.

8.3.4 Threats to validity

This section describes the main threats to validity that can affect the evaluation of our results. Given the kind of validation performed, it is worthwhile to mainly discuss threats to *construct* and *external* validity.

Threats to *construct validity* mainly concern, in this context, the measurements we performed to perform the evaluation. First, we are aware that, for assessing precision, we sampled only a subset of the extracted descriptions. However, (i) the sample size limits the estimation imprecision to $\pm 5\%$ for a confidence level of 95%, and (ii) to limit the subjectiveness and the bias in the evaluation, three evaluators (one not involved in the work and one not knowing the details of the approach) manually analyzed the sample. Another threat to construct validity concerns $\mathbf{RQ}_3$. As explained in Section 8.3.2, it is always difficult to perform a thorough assessment of *false negatives*. To deal with such a threat we evaluated a sample of 100 paragraphs not detected by the proposed heuristics. The actual number of false negatives in the entire system may be different than in the random sample.

Threats to *external validity* concern the generalization of results. We must remember that the main aim of this work is to investigate whether mailing lists and bug tracking systems are a useful source of information for understanding and potentially re-documenting source code, and to propose a novel approach to mine such descriptions (at proof of concept level), rather than to perform a thorough evaluation. The empirical evaluation here is limited to mailing lists/bug reports from two systems only. Clearly, it is important to point out that variables such as the project domain, the availability of mailing lists and bug reports (as well

Table 8.3: Examples of true positive paragraphs for Lucene.

Class	Method	Paragraph
AttributeSource	addAttributeImpl	New method added to AttributeSource: addAttributeImpl(AttributeImpl). Using reflection it walks up in the class hierarchy of the passed in object and finds all interfaces that the class or superclasses implement and that extend the Attribute interface. It then adds the interface-instance mappings to the attribute map for each of the found interfaces. AttributeImpl now has a default implementation of toString that uses reflection to print out the values of the attributes in a default formatting.
Scorer	score	This proposes to expose appropriate API on Scorer such that one can create an optimized Collector based on a given Scorer's doc-id orderness and vice versa. QueryWeight implements Weight, while score(reader) calls score(reader, false /* out-of-order */) and scorer(reader, scoreDocsInOrder) is defined abstract. One other optimization is to expose a topScorer() API (on Weight) which returns a Scorer that its score(Collector) will be called, and additionally add a start() method to DISI. That will allow Scorers to initialize either on start() or score(Collector).
Query	weight	The method Query.weight() was left in Query for backwards reasons in Lucene 2.9 when we changed Weight class. This method is only to be called on top-level queries - and this is done by IndexSearcher. This method is just a utility method, that has nothing to do with the query itsself (it just combines the createWeight method and calls the normalization afterwards). For 3.3 I will make Query.weight() simply delegate to IndexSearcher's replacement method with a big deprecation warning, so user sees this. In IndexSearcher itsself the method will be protected to only be called by itsself or subclasses of IndexSearcher.

Table 8.4: Examples of true positive paragraphs for Eclipse.

Class	Method	Paragraph
ServiceLoader	ServiceLoader	Similarly to osgi services, the java serviceloader takes the name of the class for which you want a service. In the present case, we want an instance of the JavaCompiler service, so the actual call being made is: ServiceLoader.load(javax.tool.JavaCompiler) This method returns an iterator on all the services available.
Wizard	addPages	In the particular case of the NewLocationWizard, you should be able to get around it by creating a protected createMainPage method which you can override in the subclass. You can then call super.addPages() from the subclass (Wizard) add pages to avoid the duplication of the setting of the properties. I still don't think that "alternative" is the proper term to use everywhere.
GC	drawString	The -1 value for bidiLevel is correct since it indicates that you're not using bi-directional text. As Randy mentioned, this might be a GDI+ issue that got introduced in 3.5. Create an SWT GC, invoke setAdvanced(true), and then use its drawString() method to draw some text in your language. Also try drawText().

as their quality) could influence the performances of the proposed approach. Therefore, a more extensive evaluation with data sets from further systems is highly desirable. Last but not least, the generalization of the heuristics calibration cannot be guaranteed by our evaluation.

8.4 Qualitative Analysis

This section provides a qualitative analysis of some exemplar paragraphs, identified during the manual validation. The aim is to: (i) show examples of the various kinds of descriptions that the approach is able to mine; (ii) explain why the approach, in some cases, detected false positives; and (iii) explain why the approach missed some good descriptions, i.e., false negatives. In summary, starting from what we collected during our validation, it is possible to classify the retrieved paragraphs as follows:

- *True positive paragraphs:* these are paragraphs identified by the proposed approach, that the human validation classifies as properly describing a given method. Such paragraphs can be used to help understanding the source code or to re-document it.

- *False positive paragraphs:* these are paragraphs identified by the proposed approach,

Table 8.5: Examples of false positive paragraphs for Eclipse.

Class	Method	Paragraph
Table	releaseWidget	Similar (and related) NPE is on Table class, on releaseWidget() method call - the last element in columns[] array is null.
WorkbenchPart	dispose	It must be the last method called on the contribution item. After calling dispose(), it is a bug to continue using the contribution item
OperationCanceledException	OperationCanceledException	On thinking about it, throwing OperationCanceledException would be unusual since the method does not take a progress monitor parameter. Returning a CANCEL status seems like the best approach.

however, based on the human validation, they do not really have the purpose of providing a method description. Such paragraphs reduce the precision of the approach.

- *True negative paragraphs:* these paragraphs are not selected by the proposed approach and, indeed, they do not describe methods, while they possibly refer to a method for other purposes.

- *False negative paragraphs:* these paragraphs are discarded by the proposed approach, however they represent good method descriptions.

The examples reveal several discourse patterns that characterize true positive, false positive, and false negative method descriptions. Regarding *true positives*, these paragraphs are always composed of sentences in *affirmative form*, directly explaining a method's syntax or behavior. For Lucene (Table 8.3), the first true positive example is a clear description of the *addAttributeImpl* method from the *AttributeSource* class. In this case, the developer initially informs others about the introduction of a new method and after that he explains what the method does: "finds all interfaces that the class or superclasses implement and that extend the Attribute interface" and "adds the interface or instance mappings to the attribute map for each of the found interfaces". This paragraph was extracted from a list of *candidate descriptions* with highest score (cosine=0.74), where each of these paragraph refer to 100% of the method parameters, in this case *addAttributeImpl*. We can find only phrases in *affirmative form* without sentences in *dubitative form*.

In same way, for Eclipse (Table 8.4) if we observe the first true positive example—describing the constructor *ServiceLoader*—we can find a paragraph in *affirmative form* without sentences in *dubitative form*. It is important to note that this paragraph, with respect to

Table 8.6: Examples of false positive paragraphs for Lucene.

Class	Method	Paragraph
MultiReader	isOptimized	These 3 methods should probably be fixed: isOptimized() would fail - similar to isCurrent() setNorm(int, String, float) would fail too, similar reason. directory() would not fail, but fall to return the directory of reader[0]. This is because MultiReader() (constructor) calls super with reader[0] again. I am not sure.
SegmentReader	termDocs	Yes, but this class is package private and unused! AllTermDocs is used by SegmentReader to support termDocs(null), but not AllDocsEnum. The matchAllDocs was just an example, there are more use cases, e.g. a TermsFilter (that is the non-scoring TermQuery variant): Just use the DocsEnum of this term as the DicIdSetIterator.
TopDocsCollector	topDocs	We might also consider deprecating the topDocs() methods that take in parameters and think about how the paging collector might be integrated at a lower level in the other collectors, such that one doesn't even have to think about calling a diff. collector

the first example paragraph of Lucene, obtained highest rank because if refers to 100% of the parameters of *ServiceLoader* and it contains the keywords "call" and "return" and thus it describes the method in terms of invocation of other methods and in terms of its returned value (syntactic descriptions).

If we look at false positives, for Eclipse (Table 8.5) we can notice examples of descriptions that are too specific (e.g., for the *releaseWidget* method), hence not particularly useful to properly understand the entire method. Other examples are related to faulty behavior (*dispose*) and about a possible bug fixing (constructor of *OperationCanceledException*). For Lucene (Table 8.6), the candidate description of the *isOptimized* method from the *MultiReader* class consists, actually, in a proposal of bug fixing for several methods. Regarding the *termDocs* method from the *SegmentReader* class, the paragraph mainly describes dependencies among methods rather than describing method behavior. In some sense, this could

Table 8.7: Examples of false negative paragraphs for Eclipse.

Class	Method	Paragraph
JavaStringDoubleClickSelector	doubleClicked	What it does is: - change the behavior of the doubleClicked() methods to also consider the endpoint of the mouse selection for its calculation of the text selection.
Engine	accept	If I understood TypeDescriptor.initialize() method correctly, it is not interested in the method code, so you could use ClassReader.accept(visitor, ClassReader.SKIP_CODE) to completely skip all methods code from visiting. Same applies to implementation of SearchEngine.getExtraction(..) and TagScanner.Visitor.getMethods(..) methods, where you also can add ClassReader.SKIP_CODE to avoid visiting method code.

also be considered a true positive (e.g., useful to understand method dependencies), although our evaluators classified it as a false positive because the paragraph did not clearly describe the method behavior. The last case (the *topDocs* method from the *TopDocsCollector* class) is a paragraph where people suggest to deprecate such a method and integrate the behavior elsewhere. Also in this case, the description could be, in principle, considered a useful one, although it was not considered as such because the paragraph described the behavior to be refactored. In conclusions, false positives either concern borderline cases—which could be useful in some cases and hence increase the amount of useful material a developer has to comprehend the source code—or cases such as faulty or future behavior which would not easy to discern automatically. This also suggests that the results strongly depend on the data source we use (i.e., the content of the emails and, above all, of the bug reports), indicating that some sources, such as bug reports, in some case contain descriptions that are not appropriate for describing the correct, current behavior of a method.

Finally, concerning the false negatives (Tables 8.7 and 8.8 for Eclipse and Lucene respectively), many of them were descriptions discarded because they describe the methods without containing keywords (such as, “return”, “override”, “invoke”, etc.) we used for filtering. For example, in the case of Lucene, the paragraph referring to the *optimize* method from the *IndexWriter* class contains the sentence “I found that IndexWriter.optimize(int) method does...” containing the class name and method name, yet it does not contain any of the above keywords. A similar situation occurs for the *parse* method from the *TrecFTParser* class. Similar examples can be found in Eclipse, where the *doubleClicked* method from the *JavaString-*

Table 8.8: Examples of false negative paragraphs for Lucene.

Class	Method	Paragraph
IndexWriter	optimize	I found that IndexWriter.optimize(int) method does not pick up large segments with a lot of deletes even when most of the docs are deleted. And the existence of such segments affected the query performance significantly. I created an index with 1 million docs, then went over all docs and updated a few thousand at a time. I ran optimize(20) occasionally. What saw were large segments with most of docs deleted. Although these segments did not have valid docs they remained in the directory for a very long time until more segments with comparable or bigger sizes were created.
TrecFTParser	parse	In TrecFTParser.parse(), you can extract the logic which finds the date and title into a common method which receives the strings to look for as parameters (e.g. find(String str, String start, int startlen, String end)).

DoubleClickSelector class is, again, described properly, yet none of the filtering keywords is mentioned. In conclusion, this suggests that some false negatives could have been avoided by weakening the filtering criteria, however this would also have reduced the precision and hence would have increased the amount of (possibly useless) descriptions a developer has to browse.

8.5 CODES tool: mining sourCe cOde Descriptions from developEr diScussions

This Section describe CODES (mining sourCe cOde Descriptions from developEr diScussions), an Eclipse plugin that automatically extracts Java method descriptions from SO discussions. The plugin implements a description-mining approach described in Section 8.2,

which was originally conceived to mine method descriptions from mailing lists and issue tracker discussions. The approach has been adapted to mine SO discussions too, plus it is now able to group together multiple candidate discussions for the same artifact (and remove duplicates). CODES can be used for re-documentation purposes in two different scenarios: (i) the original developers can rely online discussion to better re-document their own code, or (ii) a system integrator downloads some scarcely documented third-party source code, and needs to understand it.

Specifically, the following paragraphs describe how CODES extracts candidate method documentation from StackOverflow discussions, and creates Javadoc descriptions from it. We evaluated CODES to mine Lucene and Hibernate method descriptions. The results indicate that CODES is able to extract descriptions for 20% and 28% of the Lucene and Hibernate methods with a precision of 84% and 91% respectively.

8.5.1 Overview of the approach and its implementation in Eclipse

This section summarizes the approach behind CODES. Further details can be found in in Section 8.2 (and our previous work [13]).

Figure 8.2 depicts the CODES Eclipse plugin flow of information. First, the developer selects the list of classes that she wants to re-document. Then, Java Reflection API is used to perform an introspective analysis of the classes to be re-documented, with the aim of retrieving information about its methods, i.e., parameters, return value and the name of the method. After the introspective analysis, CODES uses the SO search engine to search the set of discussions that describe a given Java method, using as search keys project name + class name + method name. More precisely, CODES passes the search key to SO through its REST interface, i.e., through the URL[7]. Then, the SO search engine returns back the URLs of the of discussions relevant to the formulated query. Such discussions are composed of *Questions* and *Answers*.

By using regular expressions, CODES checks whether the *Questions* contain the name of the project and discards the discussions related to other projects. Note that for this purpose we do not rely on SO tags exclusively, because they are not used consistently. After that, the set of URLs is further restricted to discussions traced to the class to be re-documented. This is done using an approach inspired by the one proposed by Bacchelli *et al.* [203], i.e., it works by matching the class name (or fully qualified class name when possible) in the discussion.

Then, CODES processes paragraphs contained in the SO *Answers*. In particular, we consider answers that are voted by at least one SO user. To this end, we search for paragraphs that contain words matching names of methods of the classes to be documented. We are aware

[7] http://stackoverflow.com/search?q=search keys

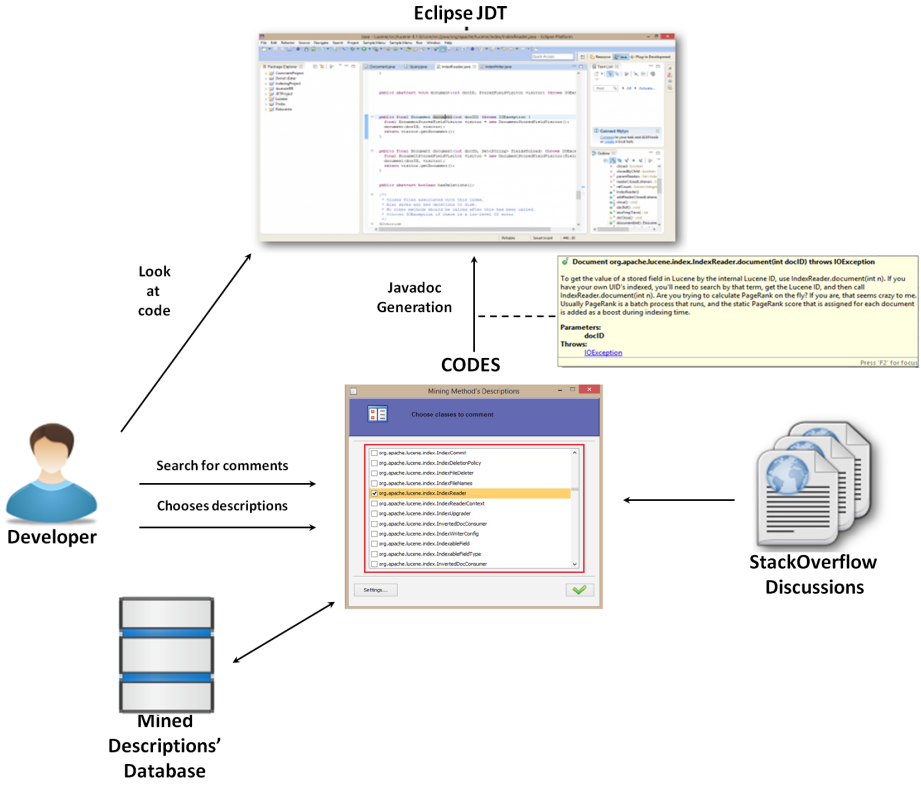
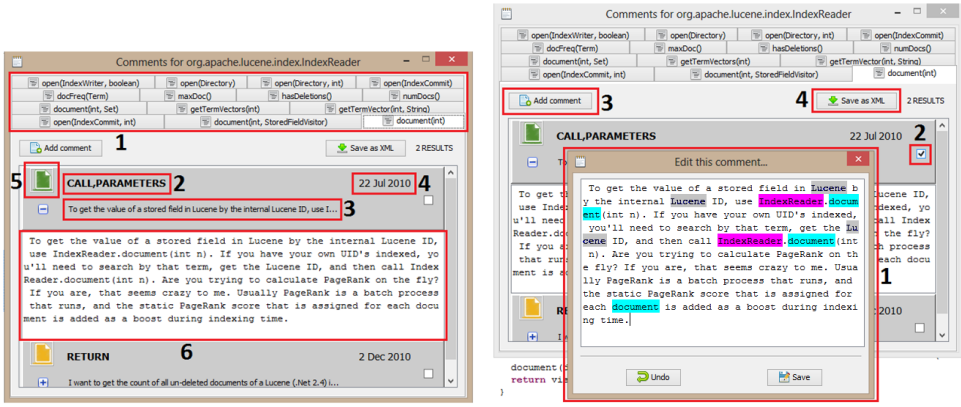


Figure 8.2: CODES information flow.

that this does not guarantee yet that the paragraph describes such a method, also because the name could have been matched by chance. After having mapped paragraphs onto methods, we classify them onto three categories of candidate method descriptions:

- *Syntactic descriptions*: this heuristic aims at identifying paragraphs that describe method syntactic descriptions. Such paragraphs contain (i) at least a given percentage s_1 of the method parameter names if the method has input parameters (i.e., paragraphs explain the method inputs, or at least some of them), and (ii) the word "return" if the method is not *void*. As explained in the previous work, we have set s_1 to 50% [13] because analyzing the distribution of parameters referred to in the paragraphs traced onto methods the median is equal to 50% for all the projects considered in that study [13].
- *Method invocations*: this heuristic aims at identifying paragraphs that describe a method behavior in terms of invocations of other methods. Such paragraphs must contains (i)



(a) Mined method Descriptions.

(b) Description editing (relevant keywords highlighted in different colors).

Figure 8.3: Browsing the mined descriptions.

words among "call", "execute", and "invoke", and (ii) must mention at least one of the methods invoked by the method being documented.

- *Overriding/Overloading*: this heuristic aims at identifying paragraphs that describe how a method overrides/overloads another. After checking if, indeed, the method being re-documented does an overload or override, we check the presence of the words "overload" or "override" in the paragraph.

Paragraphs that do not fall in any of the above three categories are discarded. Since CODES can potentially associate a discussion to more than one class, to increase the accuracy of the approach, we apply a further step, in which we compute the textual similarity between the candidate paragraphs identified in the previous step and the body of the methods they re-document. The similarity is computed by indexing both source code and textual paragraphs using a Vector Space Model [80]. After having performed stop word removal, Snowball stemming, and *tf-idf* term weighting, we compute the cosine similarity between candidate descriptions and method corpora, and discard descriptions for which the cosine is smaller than a threshold th_T (calibrated to 0.4 in our previous work [13]). After such a filtering, we use again textual similarity to prune out duplicate descriptions of the same method, i.e., paragraphs describing the same method and having a cosine similarity greater than 0.8. Finally, the obtained descriptions are stored in a local database implemented using SQLite. Then, the user can browse such descriptions from a window in the IDE, modify them, and if needed automatically inserting them in the code in form of Javadoc comments.

8.5.2 CODES in action

This section describes how CODES works, explaining its main features.

Starting to Search for Candidate Descriptions

The developer starts to use CODES by right-clicking on the Java project and selecting, in the project's context menu, the option *"Mining Method Descriptions"*. CODES opens a Window that allows the developer to select the class(es) that she wants to re-document. When the developer clicks on the "Confirm" button, CODES starts searching for method descriptions of the selected class(es). Before starting the search for candidate descriptions, the developer can click the button *"Settings"* to access in the search configuration window. This allows the developer to choose between two possible sources of descriptions: (i) a local database containing descriptions downloaded from SO during previous online searches, (ii) direct online queries to SO.

Once the developer clicks on the *"Confirm"* button, CODES starts to search for descriptions in SO and shows, with a progress bar, the status of the search process. Once the search has been completed, the developer can start browsing/analyzing the candidate descriptions found.

Browsing and Editing Candidate Descriptions

CODES generates a frame like the one shown in Figure 8.3(a) for each class for which candidate method descriptions were found in SO or in the local database. The frame contains, for each method, a tab (point 1, Figure 8.3(a)) with expandable panels, reporting the descriptions found for that method. In addition, each panel reports: (i) the description types (point 2, Figure 8.3(a)) i.e., whether it is a syntactic description, behavior description, or overload/override description, (ii) a description preview (point 3, Figure 8.3(a)), the date when the description was posted on SO (point 4, Figure 8.3(a)), and an icon (point 5, Figure 8.3(a)) colored green or yellow depending on the relevance of the description itself. CODES allows a developer to expand the panel and browse all descriptions for a given method (point 6, Figure 8.3(a)). By clicking on any of these descriptions, CODES opens a new frame (point 1, Figure 8.3(b)), which provides the possibility to edit the description text. In such a view, relevant keywords are highlighted using different colors: (i) matched method name in light blue, (ii) matched project name in gray, and (iii) matched class name in purple. From this view, the developers can also export the description(s) in XML by clicking on the *"Save as XML"* button (point 4, Figure 8.3(b)). For example, we used such option for the manual validation of the paragraphs extracted, as described in Section 8.5.3.

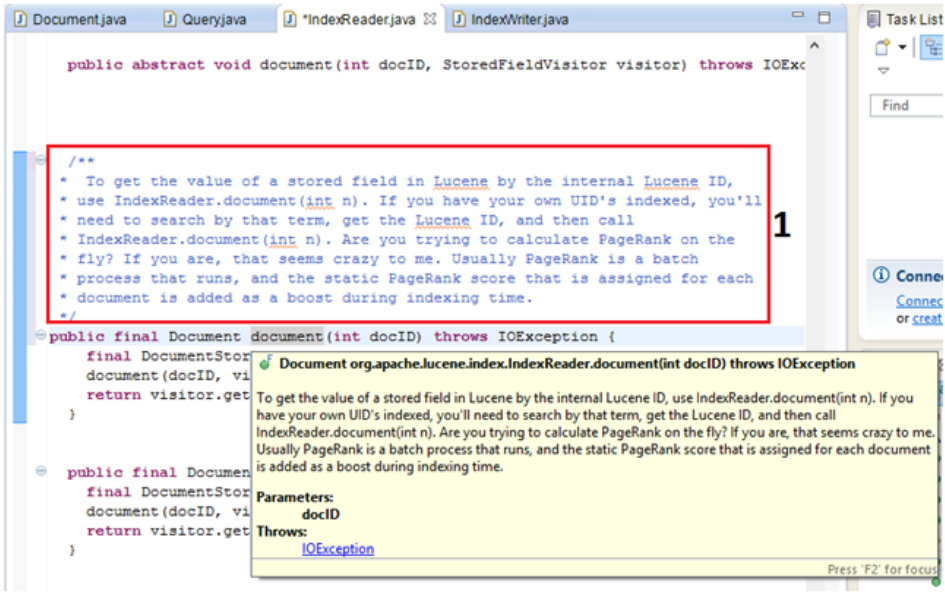


Figure 8.4: Generating a Javadoc comment from mined descriptions.

Adding Mined Descriptions as Javadoc Comments

CODES allows the developer to select (point 2, Figure 8.3(b)) one or more descriptions to build a Javadoc comment for a specific method. Once the desired descriptions, the developer can click on the *"Add Comment"* button (point 3, Figure 8.3(b)) to generate the Javadoc comment, which will be automatically inserted in the code browseable from the Eclipse-JDT IDE (point 1, Figure 8.4).

8.5.3 Performance Evaluation

In our previous study in Section 8.2([13]) we evaluated the approach implemented in CODES by mining method descriptions of Lucene and Eclipse in the issue trackers of these two projects. The proposed approach selected as good candidate method descriptions 3,111 paragraphs for Eclipse and 3,707 for Lucene, covering 36% of the methods for Lucene and 7% for Eclipse. As pointed out in our previous study, although such percentages appear to be low, they are reasonable, because it is unlikely to find, in developers' communication, a thorough description of all possible methods. A manual validation on a sample of 250 candidate descriptions for each projects indicated a precision of 79% for Eclipse and of 87% for Lucene.

To evaluate the CODES plugin when mining description from SO, we considered the

developers discussions in SO related to Lucene 2.9.0 (September 2009) and Hibernate 3.5.0 (August 2009), and applied a similar empirical evaluation that we performed in the previous work. In particular, CODES found candidate method descriptions for 20% of the Lucene's methods and 28% for the Hibernate's methods. A manual validation of 100 of the 9,343 descriptions mined for Lucene and 100 of the 10,608 descriptions mined for Hibernate indicated a precision of 84% for Lucene and 91% for Hibernate.

8.6 Related Work

Our approach relates to previous work both in its goals and execution.

Previous results that are closest to our work and used in our approach (see Section 8.2) were published by Bacchelli *et al.* [203, 204, 208]. What relates this work to our approach is the use of similar heuristics, as well as the main goal of connecting emails and source code. What differentiates our work is the emphasize on methods (rather than classes) and the specific focus on paragraphs describing the methods. Another work concerned with extracting technical information (such as, source code elements) embedded in emails and other unstructured information [209], uses spell checking tools, yet it is not concerned with identifying relevant parts of the code.

Recently Bettenburg *et al.* [210] used an approach relying on clone detection to link emails to source code. While the purpose of our work is different, their approach could potentially be used—as we plan to do in future work—to increase the performances of our approach.

Many techniques for Traceability Link Recovery (TLR) between software artifacts [211] and many recommendation systems [212] aim at connecting specific source code artifacts to unstructured text documents. TLR techniques based on text retrieval techniques [170, 201] are usually used to connect source code and text-based documentation. Two issues differentiate all such approaches to traceability from our work: (i) none of them specifically addresses methods and emails as linking artifacts; and (ii) none of them is concerned with the establishing that the external artifacts (i.e., emails in our case) contain specifically description of methods. Likewise, some recommendations systems, such as, Hipikat [213], are based on the use of text retrieval techniques. As opposed to most traceability techniques, Hipikat handles method level granularity and emails and it can recommend emails related to a method, yet it does not extract the descriptive parts of the emails.

Our approach relies on heuristics that capture discourse rules that developers follow when describing code in their communications. Previous work [214, 215] looked into how developer describe problems in bug reports. Rules of discourse in source comments were also

investigated. For example, Etzkorn *et al.* [216] found that 75% of sentence-style comments were in the past tense, with 55% being some kind of operational description (e.g., "This routine reads the data.") and 44% having the style of a definition (e.g., "General matrix"). Likewise, Tan *et al.* [217] analyzed comments written in natural language to extract implicit program rules and used these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. In the same realm, Zhong *et al.* [218] proposed an approach, called Doc2Spec, that infers resource specifications from API documentation in natural languages.

One of the potential uses of our approach is the re-documentation of source code by generating method summaries using the paragraphs extracted from the emails and bug reports. Existing work addressed the issues of using the code and comments in methods to generate method summaries. Most recent approaches used language generation techniques [219] and IR techniques [220] to generate method summaries. Some of these summaries are not unlike some of the paragraphs our approach retrieves.

8.7 Summary

We verified in this work our hypothesis that developer communications, such as, mailing lists and bug reports, contain textual information that can be extracted automatically and used to describe methods from Java source code. We found that at least 22% of the methods in Eclipse and 65% of the methods in Lucene are specifically referenced in emails and bug reports. Only a part of these references are included in paragraphs that describe the methods and can be automatically retrieved by our approach.

Our approach to mine method descriptions from developers communication—and specifically from mailing lists and bug tracking systems, first traces emails/bug reports to classes, and then, after extracting paragraphs, traces them to methods. After that, it relies on a set of heuristics to extract different kinds of descriptions, namely: (i) descriptions explaining methods in terms of their parameters and return values; (ii) descriptions explaining how a method overloads/overrides another method; and (iii) descriptions of how a method works by invoking other methods. Finally, a further pruning is performed by computing the textual similarity between the paragraphs and the methods body.

Our empirical evaluation indicates that the proposed approach is able to identify descriptions with a precision up to 79% for Eclipse and up to 87% for Lucene. The method coverage of these description is low for Eclipse, ranging between 7% and 2%, and higher for Lucene, ranging between 36% and 15%. The low method coverage is the result of two factors: (i) only part of the methods are described properly in these communications, and (ii) our ap-

proach is rather conservative as we focused on achieving high precision, given the envision usage scenario (i.e., a developer trying to understand quickly what a method does). Our investigation revealed the presence of linguistic patterns that—at least for the two analyzed systems—characterize different kinds of method descriptions.

As described in section 8.5 we implemented our approach in CODES, an Eclipse plugin to automatically extract Java method descriptions from discussions in SO. The tool is based on a "social" approach defined in our previous work [13] and adapted to mine SO. CODES searches SO for method descriptions of selected Java classes, and then recommends to the developers, who can edit such descriptions, and then ask CODES to generate Javadoc comments. The tool can be used by developers/owners of an open source project to re-document their own code or, if needed, by integrators to re-document and understand poorly commented open source code they want to reuse and integrate in their projects.

What is important to highlight is that the approach works well in both issue tracker and SO discussions, confirming the reliability of the performance of the approach behind CODES. From the other side, the methods coverage varying a lot between different projects, indeed, it depends strictly from the overall amount of discussions and the attitude of developers to report descriptions of methods in issue tracker (or in general in developers communications). However, as expected the methods coverage is lower for big systems (for example Eclipse) and higher for small projects (for example Lucene and Hibernate) but, in general the percentage value of the coverage is enough high for all the projects analyzed.

Chapter 9

Conclusions and Future Work

Contents

9.1 Summary of Contributions

This thesis investigates and explores the limits of the newcomer training process in OSS projects and suggests possible concrete solutions to help the integration of newcomers in the project evelopment team. There are various aspects that characterize a proper newcomer training: (i) project environment, (ii) newcomers expertise and known technologies, (iii) source code and documentation quality. Indeed, a proper project environment is highly desirable because helps the newcomer to approach easily with the project. Moreover, the quality and understandability of source code and the related documentation can help the newcomer to applying early changes in the code. The thesis focuses the attention in the analysis of the project environment (especially, social environment), the source code and the software documentation with the purpose to discover important facts, relevant for the improvement of the quality of the newcomer training process. Specifically, we (i) study how newcomers behave during program comprehension activities and how they interact with others developers then,

(ii) to develop tools for supporting them during development activities and in the integration in the development team.

Thus, a first part of the thesis is aimed at understanding what kind of information can be obtained by analyzing data from software repositories (e.g., versioning systems) to help newcomers to collaborate with others developers and support the team work.

Chapter 2 is aimed at understanding what kind of information can be obtained by analyzing data from versioning systems and development discussions to help newcomers to collaborate with others developers and support the team work. For this reason we analyze unstructured discussions between software developers, in form of mailing lists, issue trackers or IRC chat to extract the social interactions between project developers. We also analyze versioning systems to extract important facts about the code changes applied by experienced developers. We try to identify the more reliable communication channels to communicate with more experienced developers that cover important project roles (e.g., project coordinators). Specifically, we report a study that analyze DSN and investigate how collaboration links vary and complement each other when they are identified through data from the various kinds of communication channels. Results of a study (reported in Section 2.3) over six open source projects indicate that the overlap of communication links between the various sources is relatively low and varies between projects. This means that, the identification of key project roles for project newcomers —e.g., high degree—lead to different results when using different sources.

Chapter 3 observes how developers contributing to open source projects spontaneously group into "emerging" teams, reflected by messages exchanged over mailing lists and issue trackers. Then, it investigates how emerging teams re-organize themselves when a project evolves (e.g., by splitting or merging). Results of this study—conducted on the evolution history of four open source projects—provide indications of what happens in the project when teams reorganize. Specifically, we found that emerging team mergers and splits working on more cohesive groups of files. Such indications serve to better understand the evolution of a software project by project newcomer. More important, the observation about how emerging teams change can serve to suggest software re-modularization or re-factoring actions for newcomers/senior developers that are interested to better restructure the software components.

Chapter 4 studies the evolution of the Apache Software Ecosystem, in terms of number of developers, their interactions and the dependencies between projects and investigates (i) how dependencies between projects and the developers interactions evolve over time

when the ecosystem grows; (ii) how developers discuss the needs and risks of such upgrades. The study results suggest the a proper communication between developers belonging to different sub-projects of the ecosystem is one of the key elements that avoid the presence of bugs and/or fault, as well as, incompatibility problem between projects of the same ecosystem. Such information can be used for define an approach that help developers/newcomers to avoid changes that could break the dependency with third-party libraries.

A second part of the thesis investigates the information that can be extracted analyzing source code and the interactions between software artifacts with the main purpose to help newcomers in program comprehension task.

Chapter 5 is aimed at investigating (i) to what extent newcomers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts. Results indicate that, although newcomers spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, developers—especially more experienced ones—follow an "integrated" approach by using different kinds of artifacts. Such information can be seen as a starting point to built recommenders in help newcomer to choice appropriate patterns in navigate software documentation when apply maintenance tasks.

Chapter 6 motivates and reports an empirical study aimed at build useful code summaries of source code artifacts with the aim of facilitating newcomers comprehension. Results show that overall there is a relatively high overlap between automatic and human-generated labels, ranging between 50% and 90%. However, the highest overlap is obtained by using the simplest heuristic, while the most sophisticated techniques, i.e., LSI and LDA, provide generally the worst accuracy. Thus, the *ad-hoc* heuristics experimented in this study represent a valid approach to *build high quality summaries* of source code elements to help new developers in program comprehension.

The third part of the thesis, on the basis of insights obtained in the previous parts, presents recommenders to support concretely project newcomers.

Chapter 7 presents an approach, named YODA (Young and newcOmer Developer Assistant) aimed at identifying and recommending mentors in software projects by mining data from mailing lists, issue trackers and versioning systems. The evaluation we performed on seven software projects indicate that top committers are not always the most

appropriate mentors, and show the potential usefulness of YODA as a recommendation system to aid project managers in supporting newcomers joining a software project. Finally, we discover that a properly training by project mentors impacts the trajectory (career) of newcomers that are join the project. Well-trained newcomers have an higher permanence in the project, almost twice that of developers that do not receive any initial support.

Chapter 8 presents a recommender that mine messages exchanged among developers (or contributors), in the form of issue trackers and emails, and extract useful descriptions, that describe specific source code elements. We have evaluated the approach on bug reports and mailing lists from two open source systems (Lucene and Eclipse). The results indicate that mailing lists and bug reports contain relevant descriptions that the proposed approach identifies with a precision higher than 79%. The extracted method descriptions can help developers in understanding the code and could also be used as a starting point for source code re-documentation.

In conclusion, in this thesis we investigated problems arising when newcomers join software projects, and possible solutions to support them. After a deep analysis of software repositories we found that it is possible to support the newcomer training with various recommenders. Among the many contributions of this thesis, we support the first newcomers training stage with the suggestion of appropriate mentors; then, we help newcomers during maintenance activities improving their program comprehension with the generation of high quality source code summaries or identifying descriptions in natural language (mined from developers' discussion) describing source code elements. Clearly this thesis is only a starting point for various possible solutions/recommenders to support newcomers as well as experienced developers during comprehension and maintenance activities. Specifically, the next section outline possible future direction of this work.

9.2 Future work

Based on the work carried out in this PhD thesis, we foresee several direction for future work:

- Concerning the analysis of developers collaborations, work-in-progress aims at further validate the results by performing a survey asking to developers about the truthfulness of the social links identified by analyzing different communication channels. This information is relevant to validate the teams identified from emerging collaborations extracted from mailing lists and issue trackers.

- Concerning the analysis of the navigation patterns among software documents, we plan to explicitly investigate possible relationships existing between the way developers use the available documentation and the correctness of the tasks they perform. Also, we will aim at building recommenders to help newcomer in the choice of appropriate patterns to navigate software documentation during maintenance tasks.
- We plan to further improve the mentor recommender (YODA) by considering factors able to better capture the technical skills of mentors.
- There is still a lot of space of improvement for the method documentation miner (CODES), Specifically, we are interested to increase the precision while keeping the method coverage as high as possible, as well as reducing the percentage of false positives. Possible improvement directions include a better classification of discussion content, as well as the use of natural language parsers.
- Last, but not least, we plan to evaluate and assess the proposed recommenders in the context of user studies, and to integrate them with other kind of recommenders that can improve the newcomers training.

9.3 Replication Packages and Tools

The following sections report the replication packages and tools that available on-line to make possible to other developers replicate our analysis.

9.3.1 Replication Packages

We make available the replication packages from the above studies presented in this thesis to favor the studies replicability:

- *Chapter 2*: the replication package [1] provides: (i) the downloaded data from all sources of all projects, (ii) the developers' links extracted, and (iii) the *R* scripts and working data sets used to produce the results reported in this study.
- *Chapter 3*: the replication package[2] provides: (i) raw communication from the mailing lists and issue trackers, (ii) communication graphs obtained from mailing lists and issue trackers, (iii) parsed Git history logs, (iv) working data sets (i.e., spreadsheets) from which the statistics used to address the RQs were computed.

[1] `www.rcost.unisannio.it/mdipenta/devel-net.tgz`
[2] `www.rcost.unisannio.it/mdipenta/team-evol.tgz`

- *Chapter 4*: the replication package[3] provides information to download all analyzed projects, and includes data sets used to answer the study research questions. In particular, we provide: (i) raw data of the evolution during time of number of projects, dependencies between them, size, and number of developers of the Apache, ecosystem; (ii) the history of dependencies between project releases, and (iii) raw data of the manual tagging performed on the developers' discussions.

9.3.2 Tools

We make available the tools realized and presented in this thesis for supporting project newcomers:

- *Chapter 7*: YODA is currently available for download [4] together with a video explaining how the plugin works.
- *Chapter 8*: CODES is currently available for download [5] together with a video explaining its features through a demonstration scenario.

[3] `http://distat.unimol.it/reports/emse-apache/`

[4] `http://distat.unimol.it/tools/YODA/` or `http://www.ing.unisannio.it/spanichella/pages/projects.html`

[5] `www.ing.unisannio.it/spanichella/pages/tools/CODES`

References

[1] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multi-objective optimization problem," tech. rep., University of Salerno and University of Sannio, 2013.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," 2013.

[3] A. Lucia, M. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *Empirical Software Engineering*, pp. 1–38, 2013.

[4] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[5] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation," *Information and Software Technology*, vol. 55, no. 4, pp. 741–754, 2012.

[6] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol, "How developers' collaborations identified from different sources tell us about code changes.," in *ICSME 2014. IEEE International Conference on Software Maintenance and Evolution*, 2014.

[7] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto, "How the evolution of emerging collaborations relates to code changes: an empirical study.," in *Proceedings of the 36th International Conference on Program Comprehension*, (Hyderabad, India), pp. 177–188, 2014.

[8] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "An empirical investigation on documentation usage patterns in maintenance tasks.," in *Software Maintenance, 2013. ICSM 2013. IEEE International Conference on*, pp. 210–219, 2013.

[9] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of apache," in *Software Maintenance, 2013. ICSM 2013. IEEE International Conference on*, pp. 80–89, 2013.

[10] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, IEEE CS Press, 2013.

[11] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Cary, NC, USA), p. 44, 2012.

[12] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?," in *IEEE 20th International Conference on Program Comprehension (ICPC'12)*, (Passau, Germany, June 11-13), pp. 193–202, 2012.

[13] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *IEEE 20th International Conference on Program Comprehension, ICPC 2012*, pp. 63–72, 2012.

[14] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery using smoothing filters," in *Proceedings of the 19th IEEE International Conference on Program Comprehension*, pp. 21–30, 2011.

[15] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, and G. Canfora, "Recommending refactorings based on team co-maintenance patterns," in *Proceedings of the 29th international conference on Automated Software Engineering (ASE 2014)*, 2014.

[16] G. C. D. P. M. C. Vassallo, S. Panichella, "Codes: mining source code descriptions from developers discussions.," in *Proceedings of the 36th International Conference on Program Comprehension*, (Hyderabad, India), pp. 106–109, 2014.

[17] G. Canfora, M. Di Penta, S. Giannantonio, R. Oliveto, and S. Panichella, "Yoda: Young and newcomer developer assistant," in *Proceedings of the 35th International Conference on Software Engineering*, IEEE CS Press, 2013.

[18] I. Steinmacher, I. Wiese, A. Chaves, and M. Gerosa, "Newcomers withdrawal in open source software projects: Analysis of hadoop common project," in *Collaborative Systems (SBSC), 2012 Brazilian Symposium on*, pp. 65–74, 2012.

[19] I. Steinmacher, I. Wiese, A. Chaves, and M. Gerosa, "Why do newcomers abandon open source software projects?," in *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pp. 25–32, 2013.

[20] I. Steinmacher, I. S. Wiese, T. Conte, M. A. Gerosa, and D. Redmiles, "The hard life of open source software project newcomers," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pp. 72–78, ACM, 2014.

[21] R. Kraut, M. Burke, and J. Riedl, "Dealing with newcomers," 2010.

[22] M. Zhou and A. Mockus, "Does the initial environment impact the future of developers?," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pp. 271–280, ACM, 2011.

[23] A. Hars and S. Ou, "Working for free? motivations for participating in open-source projects," *Int. J. Electron. Commerce*, vol. 6, no. 3, pp. 25–39, 2002.

[24] S. Krishnamurthy, "On the intrinsic and extrinsic motivation of free/libre/open source (floss) developers," *Knowledge, Technology & Policy*, vol. 18, no. 4, pp. 17–39, 2006.

[25] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. de Vries, "Moving into a new software project landscape," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pp. 275–284, ACM, 2010.

[26] E. Capra and A. I. Wasserman, "A framework for evaluating managerial styles in open source projects," in *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, OSS 2008, September 7-10, 2008, Milano, Italy*, pp. 1–14, Springer, 2008.

[27] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, (New York, NY, USA), pp. 137–143, ACM, 2006.

[28] E. Shihab, Z. M. Jiang, and A. Hassan, "Studying the use of developer irc meetings in open source projects," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 147–156, 2009.

[29] K. Ehrlich and N. S. Shami, "Microblogging inside and outside the workplace," in *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010*, The AAAI Press, 2010.

[30] L. Singer, F. Figueira Filho, and M.-A. Storey, "Software engineering at the speed of light: How developers stay current using twitter," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 211–221, ACM, 2014.

[31] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pp. 277–286, IEEE / ACM, 2013.

[32] I. Steinmacher, I. Wiese, and M. Gerosa, "Recommending mentors to software project newcomers," in *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pp. 63–67.

[33] F. Brooks, *The Mythical Man-Month 20th anniversary edition*. Boston, MA, USA: Addison-Wesley, 1995.

[34] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 298–308, 2009.

[35] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, (New York, NY, USA), pp. 24–35, ACM, 2008.

[36] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pp. 323–332, IEEE, 2011.

[37] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *International Conference on Program Comprehension, ICPC 2010*, pp. 124–133, 2010.

[38] A. Kumar and A. Gupta, "Evolution of developer social network and its impact on bug fixing process," in *Proceedings of the 6th India Software Engineering Conference*, pp. 63–72, ACM, 2013.

[39] A. Meneely and L. Williams, "Socio-technical developer networks: Should we trust our measurements?," in *Proceedings of the 33rd International Conference on Software Engineering*, (New York, NY, USA), pp. 281–290, ACM, 2011.

[40] M. Pohl and S. Diehl, "What dynamic network metrics can tell us about developer roles," in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, CHASE '08, (New York, NY, USA), pp. 81–84, ACM, 2008.

[41] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, March 2003.

[42] M. Nikulin, "Hellinger distance," *Encyclopedia of Mathematics*, 2001.

[43] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th IEEE/ACM International Conference on Software Engineering, ICSE 2013*, (San Francisco, CA, USA, May 18-26), pp. 522–531, 2013.

[44] J. P. Scott, *Social Network Analysis: A Handbook (2nd edition)*. Sage Publications Ltd, 2000.

[45] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pp. 143–152, 2011.

[46] M.-A. S. Leif Singer, Fernando Figueira Filho, "Software engineering at the speed of light: How developers stay current using Twitter," in *36sth International Conference on Software Engineering, ICSE 2014, May 31- June 07, 2009, Hyderabad, India*, 2014.

[47] A. Capiluppi and M. Michlmayr, "From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects," in *Open Source Development, Adoption and Innovation*, pp. 31–44, International Federation for Information Processing, Springer, 2007.

[48] L. Lopez, J. M. Gonzalez-Barahona, and G. Robles, "Applying social network analysis to the information in CVS repositories," in *Proceedings of the International Workshop on Mining Software Repositories*, ACM Press, 2004.

[49] A. Meneely, M. Corcoran, and L. Williams, "Improving developer activity metrics with issue tracking annotations," in *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '10, pp. 75–80, ACM, 2010.

[50] P. V. Singh, "The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 2, 2010.

[51] D. Surian, D. Lo, and E.-P. Lim, "Mining collaboration patterns from a large developer network," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pp. 269–273, 2010.

[52] J. Xu, Y. Gao, S. Christley, and G. Madey, "A topological analysis of the open source software development community," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pp. 198.1–, IEEE Computer Society, 2005.

[53] L. Yu and S. Ramaswamy, "Mining CVS repositories to understand open-source project developer roles," in *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pp. 8–8, 2007.

[54] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 375–385, IEEE Press, 2012.

[55] E. Shihab, N. Bettenburg, B. Adams, and A. E. Hassan, "On the central role of mailing lists in open source projects: An exploratory study," in *Proceedings of the 2009 International Conference on New Frontiers in Artificial Intelligence*, (Berlin, Heidelberg), pp. 91–103, Springer-Verlag, 2010.

[56] P. Wagstrom, J. Herbsleb, and K. Carley, "A social network approach to free/open source software simulation," in *Proceedings of the 1st International Conference on Open Source Systems*, (Genova, Italy), 2005.

[57] E. Shihab, Z. M. Jiang, and A. E. Hassan, "On the use of internet relay chat (IRC) meetings by developers of the GNOME GTK+ project," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 107–110, IEEE Computer Society, 2009.

[58] M. S. Elliott and W. Scacchi, "Free software developers as an occupational community: Resolving conflicts and fostering collaboration," in *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '03, pp. 21–30, ACM, 2003.

[59] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 492–501, ACM, 2006.

[60] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla," in *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 139–148, IEEE Computer Society, 2012.

[61] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, vol. 10, no. 2, 2005.

[62] C. Haythornthwaite, "The strength and the impact of new media," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 1 - Volume 1*, pp. 1019–, IEEE Computer Society, 2001.

[63] D. Zhao and M. B. Rosson, "How and why people twitter: The role that micro-blogging plays in informal communication at work," in *Proceedings of the ACM 2009 International Conference on Supporting Group Work*, pp. 243–252, ACM, 2009.

[64] J. Zhang, Y. Qu, J. Cody, and Y. Wu, "A case study of micro-blogging in the enterprise: Use, value, and related issues," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 123–132, ACM, 2010.

[65] K. Dullemond, B. v. Gameren, M.-A. Storey, and A. v. Deursen, "Fixing the "out of sight out of mind" problem: One year of mood-based microblogging in a distributed software team," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pp. 267–276, IEEE Press, 2013.

[66] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 125–134, ACM, 2010.

[67] E. S. Raymond, *The Cathedral and the Bazaar*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1st ed., 1999.

[68] C. Bird, D. S. Pattison, R. M. D'Souza, V. Filkov, and P. T. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pp. 24–35, ACM, 2008.

[69] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pp. 277–286, IEEE / ACM, 2013.

[70] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: Its extracted software architecture," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pp. 555–563, ACM, 1999.

[71] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 692–701, IEEE / ACM, 2013.

[72] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pp. 137–143, 2006.

[73] J. Bezdek, *Pattern recognition with fuzzy objective function algorithms*. New York: Plenum, 1981.

[74] R Core Team, *R: A Language and Environment for Statistical Computing*. 2012. ISBN 3-900051-07-0.

[75] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, 2005.

[76] A. Gordon, *Classification (2nd edition)*. CRC Press, 1988.

[77] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IWPC*, pp. 45–, IEEE Computer Society.

[78] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[79] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering (EMSE)*, vol. 14, no. 1, pp. 5–32, 2009.

[80] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[81] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 3rd edition ed., 1998.

[82] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition ed., 2005.

[83] J. Cohen, *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition ed., 1988.

[84] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 125–134, ACM.

[85] D. Surian, D. Lo, and E.-P. Lim, "Mining collaboration patterns from a large developer network," *Reverse Engineering, Working Conference on*, pp. 269–273, 2010.

[86] J. Bosh, "From software product lines to software ecosystems," in *Proceedings of the 13th International Conference on Software Product Lines (SPLC)*, pp. 111–119, 2009.

[87] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A sense of community: A research agenda for software ecosystems," in *31st International Conference on Software Ecosystems, New and Emerging Research Track*, pp. 187–190, 2005.

[88] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: the case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, (New York, NY, USA), pp. 56:1–56:11, ACM, 2012.

[89] M. Di Penta, D. M. Germán, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the evolution of software licensing," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pp. 145–154, ACM, 2010.

[90] J. Businge, A. Serebrenik, and M. van den Brand, "Survival of Eclipse third-party plug-ins," in *28th IEEE International Conference on Software Maintenance (ICSM 2012), Trento, Italy, Sep 23-28, 2012*, pp. 368–377, IEEE Computer Society, 2012.

[91] D. German, B. Adams, and A. E. Hassan, "Programming Language Ecosystems: the Evolution of R," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, (Genova, Italy), pp. 243–252, 2013.

[92] G. Bavota, A. Ciemniewska, I. Chulani, A. De Nigro, M. Di Penta, D. Galletti, R. Galoppini, T. F. Gordon, P. Kedziora, I. Lener, F. Torelli, R. Pratola, J. Pukacki, Y. Rebahi, and S. G. Villalonga, "The market for open source: An intelligent virtual open source marketplace," in *Joint 18th European Conference on Software Maintenance and Reengineering / 21st Working Conference on Reverse Engineering, CSMR18/WCRE21, February 3-6, 2014, Antwerp, Belgium, Proceedings*, p. To appear, 2014.

[93] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–716, 1966.

[94] M. Goeminne, M. Claes, and T. Mens, "A historical dataset for the GNOME ecosystem," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 225–228, IEEE Press, 2013.

[95] S. Gala-Perez, G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Intensive metrics for the study of the Evolution of Open Source Projects," in *10th IEEE Working Conference on Mining Software Repositories*, (San Francisco, California, USA), 2013.

[96] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, (Washington, DC, USA), pp. 131–140, IEEE Computer Society, 2000.

[97] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 262–285, 2009.

[98] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running inter-dependencies of software," in *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, (Washington, DC, USA), pp. 140–149, IEEE Computer Society, 2007.

[99] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *In Proceedings of ASE 2010*, pp. 309–312, ACM Society Press, 2010.

[100] M. Wermelinger and Y. Yu, "Analyzing the evolution of Eclipse plugins," in *Proceedings of the 2008 international working conference on Mining software repositories*, (New York, NY, USA), pp. 133–136, ACM, 2008.

[101] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi, "Assessing architectural evolution: a case study.," *Empirical Software Engineering*, vol. 16, no. 5, pp. 623–666.

[102] T. Mens, J. Fernández-Ramil, and S. Degrandsart, "The evolution of Eclipse," in *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pp. 386–395, IEEE, 2008.

[103] Y. H. Kidane and P. A. Gloor, "Correlating temporal communication patterns of the Eclipse open source community with performance and creativity," *Comput. Math. Organ. Theory*, vol. 13, pp. 17–27, Mar. 2007.

[104] D. M. Germán, "The GNOME project: a case study of open source, global software development," *Software Process: Improvement and Practice*, vol. 8, no. 4, pp. 201–215, 2003.

[105] M. Goeminne and T. Mens, "Analyzing ecosystems for open source software developer communities," in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry* (M. A. C. Slinger Jansen, Sjaak Brinkkemper, ed.), pp. 301–329, Edward Elgar Publishing, Incorporated, 2013.

[106] S. Koch and G. Schneider, "Effort, cooperation and coordination in an open source software project: GNOME," *Information Systems Journal*, vol. 12, no. 1, pp. 27–42, 2002.

[107] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workloadâĂŤ A case study of the Gnome ecosystem community," *Empirical Software Engineering*, pp. 1–54, 2013.

[108] C. Jergensen, A. Sarma, and P. Wagstrom, "The onion patch: migration in open source ecosystems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, (New York, NY, USA), pp. 70–80, ACM, 2011.

[109] T. Kilamo, I. Hammouda, T. Mikkonen, and T. Aaltonen, "From proprietary to open sourceâĂŤgrowing an open source ecosystem," *Journal of Systems and Software*, vol. 85, no. 7, pp. 1467 – 1478, 2012.

[110] W. Scacchi and T. A. Alspaugh, "Understanding the role of licenses and evolution in open architecture software ecosystems," *Journal of Systems and Software*, vol. 85, pp. 1479–1494, July 2012.

[111] J. Ossher, S. K. Bajracharya, and C. V. Lopes, "Automated dependency resolution for open source software," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pp. 130–140, IEEE, 2010.

[112] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between GNOME projects," in *2010 7th IEEE Working Conference on Mining Software Repositories, MSR 2010* (J. Whitehead and T. Zimmermann, eds.), pp. 98–101, IEEE.

[113] L. Yu, S. Ramaswamy, and J. Bush, "Software evolvability: An ecosystem Point of View," *IEEE International Workshop on Software Evolvability*, vol. 0, pp. 75–80, 2007.

[114] M. Annosi, M. Di Penta, and G. Tortora, "Managing and assessing the risk of component upgrades," in *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pp. 9–12, 2012.

[115] A. von Mayrhauser and A. M. Vans, "Comprehension processes during large scale maintenance," in *Proceedings of the 16th international conference on Software engineering*, ICSE '94, (Los Alamitos, CA, USA), pp. 39–48, IEEE Computer Society Press, 1994.

[116] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: An experimental evaluation," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 365–381, 2006.

[117] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Transaction on Software Engineering*, vol. 34, no. 3, pp. 407–432, 2008.

[118] E. Tryggeseth, "Report from an experiment: Impact of documentation on maintenance," *Empirical Software Engineering*, vol. 2, no. 2, pp. 201–207, 1997.

[119] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini, "Evaluating the effect of composite states on the understandability of UML statechart diagrams," in *proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS 2005)*, Springer, 2005.

[120] M. C. Otero and J. J. Dolado, "An initial experimental assessment of the dynamic modelling in UML," *Empirical Software Engineering*, vol. 7, no. 1, pp. 27–47, 2002.

[121] S. Tilley and S. Huang, "A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding," in *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, (New York, NY, USA), pp. 184–191, ACM Press, 2003.

[122] B. de Alwis and G. C. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, (Brighton, UK), pp. 51–54, IEEE Computer Society, 2006.

[123] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Oregon, USA), pp. 1–11, ACM, 2006.

[124] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. London: Pinter, 1992.

[125] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 96–118, 2010.

[126] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius, "Empirical evidence about the UML: a systematic literature review," *Software: Practise and Experience*, vol. 41, no. 4, pp. 363–392, 2011.

[127] M. Torchiano, "Empirical assessment of UML static object diagrams," in *International Workshop on Program Comprehension (IWPC 2004)*, pp. 226–229, IEEE Computer Society, 2004.

[128] L. C. Briand, Y. Labiche, M. Di Penta, and H. D. Yan-Bondoc, "An experimental investigation of formality in UML-based development," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 833–849, 2005.

[129] S. Abrahão, C. Gravino, E. Insfrán, G. Scanniello, and G. Tortora, "Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments," *IEEE Transaction on Software Engineering*, vol. 39, no. 3, pp. 327–342, 2013.

[130] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[131] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[132] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research*, (Toronto, Ontario, Canada), p. 21, IBM, 1997.

[133] R. DeLine, A. Khella, M. Czerwinski, and G. G. Robertson, "Towards understanding programs through wear-based filtering," in *Proceedings of the ACM 2005 Symposium on Software Visualization*, (St. Louis, Missouri, USA), pp. 183–192, ACM, 2005.

[134] M.-A. D. Storey, K. Wong, and H. A. Müller, "How do program understanding tools affect how programmers understand programs?," *Science of Computer Programming*, vol. 36, no. 2-3, pp. 183–207, 2000.

[135] T. Fritz, G. C. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Dubrovnik, Croatia), pp. 341–350, ACM, 2007.

[136] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?," *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[137] Y.-G. Guéhéneuc, "TAUPE: towards understanding program comprehension," in *Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006), October 16-19, 2006, Toronto, Ontario, Canada*, pp. 1–13, IBM, 2006.

[138] S. Yusuf, H. H. Kagdi, and J. I. Maletic, "Assessing the comprehension of UML class diagrams via eye tracking," in *Proceedings of the 15th International Conference on Program Comprehension*, (Banff, Alberta, Canada), pp. 113–122, IEEE Computer Society, 2007.

[139] B. Sharif and J. I. Maletic, "An eye tracking study on the effects of layout in understanding the role of design patterns," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, (Timisoara, Romania), pp. 1–10, IEEE Computer Society, 2010.

[140] S. Jeanmart, Y.-G. Guéhéneuc, H. A. Sahraoui, and N. Habra, "Impact of the visitor pattern on program comprehension and maintenance," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, (Lake Buena Vista, Florida, USA), pp. 69–78, 2009.

[141] V. Lavrenko, *A Generative Theory of Relevance*, vol. 26. Springer, 2009.

[142] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension*, (Paris, France), pp. 271–280, IEEE Computer Society, 2002.

[143] M.-A. D. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[144] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, (Shanghai, China), pp. 492–501, ACM Press, 2006.

[145] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[146] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 17th Working Conference on Reverse Engineering*, (Beverly, MA, USA), pp. 35–44, IEEE Computer Society, 2010.

[147] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[148] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, pp. 55–64, IEEE Computer Society, 2010.

[149] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Codetopics: which topic am i coding now?," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pp. 1034–1036, ACM, 2011.

[150] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?," in *Proceedings of the 28th International Conference on Software Maintenance*, (Riva del Garda, Italy), IEEE CS Press, 2012.

[151] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers," *Journal of Software Maintenance*, vol. 19, no. 4, pp. 205–229, 2007.

[152] A. Takang, P. Grubb, and R. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[153] 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*, vol. 41, no. 6, pp. 391–407, 1990.

[154] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, vol. 1, ch. Real rectangular matrices. Boston: Birkhauser, 1998.

[155] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[156] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[157] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (Cape Town, South Africa), pp. 223–226, ACM Press, 2010.

[158] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[159] R. D. Baker, "Modern permutation test software," in *Randomization Tests* (E. Edgington, ed.), Marcel Decker, 1995.

[160] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 625–56, 1948.

[161] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transaction on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[162] L. Guerrouj, M. D. Penta, G. Antoniol, and Y. G. Guéhéneuc, "Tidier: An identifier splitting approach using speech recognition techniques," *Journal of Software Evolution and Processes*, p. 31, 2011.

[163] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of 11th IEEE International Symposium on Software Metrics*, (Como, Italy), pp. 20–29, IEEE CS Press, 2005.

[164] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings of 16th IEEE International Conference on Automated Software Engineering*, (San Diego, California, USA), pp. 107–114, IEEE CS Press, 2001.

[165] D. Poshyvanyk, Y. Gael-Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[166] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pp. 469 – 478, 2006.

[167] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 2, no. 37, pp. 205–227, 2011.

[168] D. Binkley, H. Feild, D. Lawrie, and M. Pighin, "Software fault prediction using language processing," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, pp. 99–110, IEEE Computer Society, 2007.

[169] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[170] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of 25th International Conference on Software Engineering*, (Portland, Oregon, USA), pp. 125–135, IEEE CS Press, 2003.

[171] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artefact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, 2007.

[172] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods.," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[173] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *Proc. of ICSM*, pp. 133–142, 2011.

[174] H. U. Asuncion, A. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (Cape Town, South Africa), pp. 95–104, ACM Press, 2010.

[175] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proc. of ICSE*, pp. 155–164, 2010.

[176] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of 23rd International Conference on Software Engineering*, (Toronto, Ontario, Canada), pp. 103–112, IEEE CS Press, 2001.

[177] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[178] P. Baldi, C. V. Lopes, E. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (Nashville, TN, USA), pp. 543–562, ACM Press, 2008.

[179] E. Linstead, C. V. Lopes, and P. Baldi, "An application of latent dirichlet allocation to analyzing software evolution," in *Proceedings of the 7th International Conference on Machine Learning and Applications*, (San Diego, California, USA), pp. 813–818, IEEE CS Press, 2008.

[180] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Modeling the evolution of topics in source code histories," in *Proceedings of the 8th International Working Conference on Mining Software Repositories*, (Honolulu, HI, USA), pp. 173–182, IEEE Press, 2011.

[181] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of software maintenance activities," in *Proceedings of*

the 8th International Working Conference on Mining Software Repositories, (Waikiki, Honolulu, USA), pp. 163–172, IEEE CS Press, 2011.

[182] S. Medini, G. Antoniol, Y.-G. Guéhéneuc, M. Di Penta, and P. Tonella, "Scan: an approach to label and relate execution trace segments," in *Proceedings of the 19th Working Conference on Reverse Engineering*, (Kingston, Ontario, Canada), IEEE Press, 2012.

[183] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (Cape Town, South Africa), pp. 505–514, ACM Press, 2010.

[184] S. Rastkar, "Summarizing software concerns," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Student research competition*, (Cape Town, South Africa), pp. 527–528, ACM Press, 2010.

[185] R. P. L. Buse and W. Weimer, "Automatically documenting program changes," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, (Antwerp, Belgium), pp. 33–42, ACM Press, 2010.

[186] G. Murphy, *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[187] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, (Antwerp, Belgium), pp. 43–52, ACM Press, 2010.

[188] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, (Honolulu, HI, USA), pp. 101–110, ACM Press, 2011.

[189] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, p. 10, 2011.

[190] C. Wang, J. Han, Y. Jia, J. Tang, D. Zhang, Y. Yu, and J. Guo, "Mining advisor-advisee relationships from research publication networks," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pp. 203–212, 2010.

[191] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pp. 1767–1772, ACM, 2006.

[192] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, Szeged, Hungary, September 5-9, 2011*, pp. 365–375, ACM, 2011.

[193] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, Szeged, Hungary, September 5-9, 2011*, pp. 4–14, ACM, 2011.

[194] G. Robles, J. M. González-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, Vancouver, BC, Canada, May 16-17, 2009*, pp. 167–170, IEEE, 2009.

[195] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 ed., 2007.

[196] I. Fronza, A. Sillitti, and G. Succi, "An interpretation of the results of the analysis of pair programming during novices integration in a team," in *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*, pp. 225–235, 2009.

[197] M. Zhou and A. Mockus, "Growth of newcomer competence: challenges of globalization," in *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pp. 443–448, 2010.

[198] V. S. Sinha, S. Mani, and S. Sinha, "Entering the circle of trust: developer initiation as committers in open-source projects," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pp. 133–142, IEEE, 2011.

[199] C. Bird, A. Gourley, P. T. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," in *Fourth International Workshop on Mining Software Repositories, MSR 2007, Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, p. 6, IEEE Computer Society, 2007.

[200] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 446–465, 2005.

[201] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[202] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 205–214, IEEE Computer Society, 2009.

[203] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pp. 375–384, ACM, 2010.

[204] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, pp. 24–33, IEEE Computer Society, 2010.

[205] D. Klein and C. D. Manning, "Fast exact inference with a factored model for natural language parsing," in *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, pp. 3–10, MIT Press, 2002.

[206] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pp. 134–143, IEEE Computer Society, 2003.

[207] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[208] A. Bacchelli, M. Lanza, and V. Humpa, "RTFM (read the factual mails) - augmenting program comprehension with remail," in *Proceedings of the European Conference on Software Maintenance and Reenginering*, pp. 15–24, 2011.

[209] N. Bettenburg, B. Adams, A. E. Hassan, and M. Smidt, "A lightweight approach to uncover technical artifacts in unstructured data," in *Proceedings of the IEEE International Conference on Program Comprehension*, pp. 185–188, 2011.

[210] N. Bettenburg, S. W. Thomas, and A. E. Hassan, "Using fuzzy code search to link code fragments in discussions to source code," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pp. 319–328, IEEE, 2012.

[211] J. Cleland-Huang, O. Gotel, and A. E. Zisman, *Software and Systems Traceability*. Springer, February 2012.

[212] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, pp. 80–86, July/August 2010.

[213] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, 2005.

[214] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 127–134, 2006.

[215] A. J. Ko and P. K. Chilana, "Design, discussion, and dissent in open bug reports," in *iConference*, pp. 106–113, 2011.

[216] L. H. Etzkorn, L. L. Bowen, and C. G. Davis, "An approach to program understanding by natural language understanding," *Natural Language Engineering*, vol. 5, pp. 1–18, 1999.

[217] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or bad comments? */," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[218] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring specifications for resources from natural language API documentation," *Automated Software Engineering Journal*, 2011.

[219] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 43–52, 2010.

[220] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the IEEE International Working Conference on Reverse Engineering*, pp. 35–44, 2010.

List of Figures

List of Tables

Acronyms

ANOVA Analysis of Variance

API Application Programming Interface

ASF Apache Software Foundation

CCBC Conceptual Coupling Between Classes

CCM Conceptual Coupling Between Methods

CODES mining sourCe cOde Descriptions from developErs diScussions

DOAP Description Of A Project

DSN Developers' Social Network

FLUORITE Full of Low−level User Operations Recorded In The Editor

HLA High-Level Artifacts

IDE Interactive Development Environment

IR Information Retrieval

IRC Internet Relay Chat

KLOC Kilo Lines of Code

KNLOC Non-commented Kilo Lines of Code

LDA latent Dirichlet allocation

LLA Low-Level Artifacts

LSI Latent Semantic Indexing

ML Machine Learning

MQ Modularization Quality

OSS Open-source Software

REST Representational State Transfer

RTM Relational Topics Model

SE Software Engineering

SNA Social Network Analysis

SVD Singular Value Decomposition

SVN version control system

TLR Traceability Link Recovery

TPL Third-Party Libraries

UML Unified Modeling Language