Software Documentation Automation and Challenges

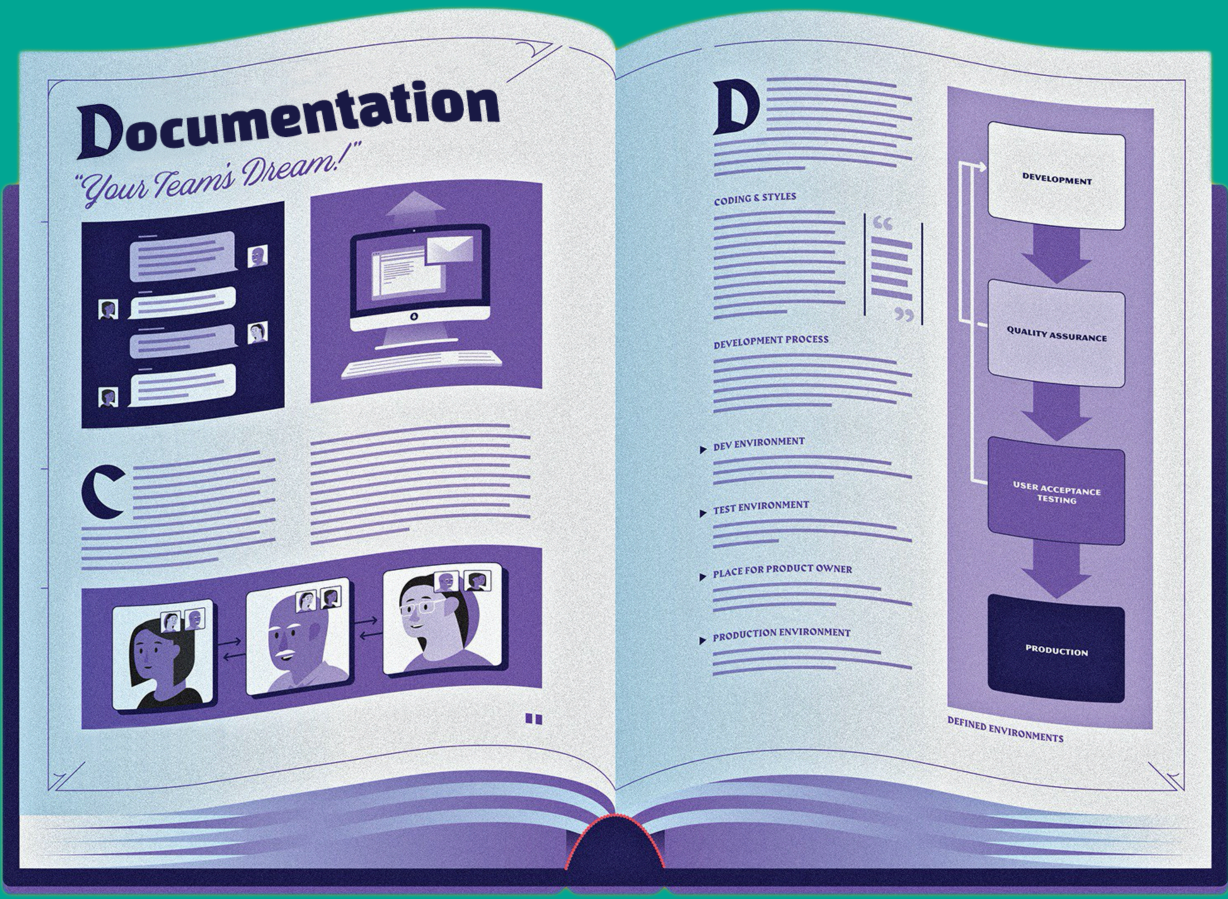
Emad Aghajani

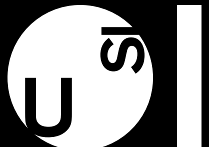
Research Advisor

Prof. Dr. Michele Lanza

Research Co-Advisor

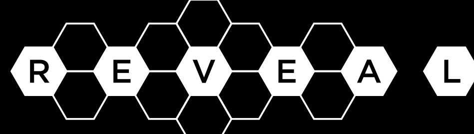
Prof. Dr. Gabriele Bavota





Università della Svizzera italiana

Software Institute



Dissertation Committee

Cesare Pautasso	Università della Svizzera italiana, Switzerland
Paolo Tonella	Università della Svizzera italiana, Switzerland
Anthony Cleve	University of Namur, Belgium
Nicole Novielli	University of Bari, Italy

Dissertation accepted on 18 June 2020

Research Advisor	Co-Advisor
Prof. Dr. Michele Lanza	**Prof. Dr. Gabriele Bavota**
Ph.D. Program Co-Director	Ph.D. Program Co-Director
Prof. Walter Binder	**Prof. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Emad Aghajani
Lugano, 18 June 2020

To my family, with love and gratitude.

Abstract

DESPITE the undeniable practical benefits of documentation during software development and evolution activities, its creation and maintenance is often neglected, leading to inadequate and even inexistent documentation. Thus, it is not unusual for developers to deal with unfamiliar code they have difficulties in comprehending. Browsing the official documentation, or accessing online resources, such as Stack Overflow, can help in this "code comprehension" activity that, however, remains highly time-consuming.

Enhancing the code comprehension process has been the goal of several works aimed at automatically documenting software artifacts. Although these techniques addressed the issue, they exhibit a number of major limitations such as working at a coarse-grained level, and not allowing to document a single line of code of interest. While the creation of such novel systems entails conceptual and technical challenges related to the collection, inference, interpretation, selection, and presentation of useful information, it also requires solid empirical foundations on software developers' needs — *what* information is (or is not) useful *when* to developers.

Our thesis is that *empirical knowledge about software documentation issues experienced and considered relevant by practitioners is instrumental to lay the foundations for the next-generation tools and techniques for automated software documentation*.

To this aim, in this dissertation we present our research accomplishments towards automating developer documentation on two fronts: (1) empirical studies on the nature of software documentation with a specific focus on documentation issues experienced by software developers, and (2) development of tools supporting the code comprehension process.

In the former direction, we conducted a large-scale empirical study, where we mined, analyzed, and categorized a large number of documentation-related artifacts and developed a detailed taxonomy of documentation issues from which we infer a series of actionable proposals both for researchers and practitioners. We validated our findings by surveying professional software practitioners. In the latter direction, we developed ADANA, a framework which generates fine-grained code comments for a given piece of code at the granularity level intended by the developer.

Our contributions to the body of software documentation knowledge shed light on unseen facts about overlooked software documentation matter and lay the foundations for the next-generation tools and techniques for automated software documentation.

Acknowledgements

Back in 2016, when I decided to pursue my studies further with a Ph.D., I could have never imagined how that could change the person I was. The Ph.D. for me was not just 4 more years of attending university classes, teaching and finally adding another row in the education section of my resume. The Ph.D. has provided me the unique opportunity to learn how to live by myself, encounter problems, and struggle to solve them, like never before. The opportunity to meet new people with different mindsets and thoughts was only one of the many priceless lessons I learned throughout these years which gave me a whole new perspective on life.

I would like to express my special appreciation and thanks to my fantastic advisors Prof. Michele Lanza and Prof. Gabriele Bavota for having given me such an opportunity and trusting me in the first place. You have been tremendous mentors for me. Beyond that, Michele with years of experience, and Gabriele with incredible perseverance provided the perfect success environment which was truly unrivaled. From the bottom of my heart, thank you for encouraging my research with your boundless support. I never forget the priceless opportunity you have given to me.

I also would like to express my gratitude to all people who specifically supported me during my Ph.D. journey. Particularly, I would like to thank all my colleagues and friends in the REVEAL research group, my second home. Andrea Mocci, Luca Ponzanelli, Roberto Minelli, Tommaso Dal Sasso, Bin Lin, Jevgenija Pantiuchina, Csaba Nagy, Alejandro Mazuera Rozo, and Fengcai Wen. You made each of my working day special and truly enjoyed every moment with you. Meeting all you great people has been a real blessing.

I would like to specifically thank my colleague Andrea for supporting me through the early days of my journey with all its difficulties. Thanks to Bin and Jevgenija who did not let me feel alone in this journey. A huge thanks to Roberto for his countless helps with the Ph.D. life. And my special thanks go to Csaba who never left me alone and stood by me in the most difficult times: you taught me a lot on how to be a better researcher.

Many of the achievements I obtained during my Ph.D. were not possible without collaboration with other great people. During my Ph.D. I had the opportunity and pleasure to collaborate with Prof. Mario Linares-Vásquez, Prof. Laura Moreno, Prof. David C. Shepherd, Prof. Anthony Cleve, Prof. Emad Shihab, and Olga Lucero Vega-Márquez. Many thanks for the great opportunity you offered me.

Above all, I should thank my family. Words can not express how grateful I am to my mother and father for all of the sacrifices that you have made on my behalf. Your prayer for me was what sustained me thus far. Also thanks my brother, Mohammad, to be there when I needed him.

I would also like to thank my beloved wife, Maliheh. Thank you for supporting me for everything and supporting me patiently while I was writing this thesis. I can't thank you enough for encouraging me throughout this experience.

Finally, I thank my God for letting me through all the difficulties. I have experienced Your support day by day. I will keep on trusting You for my future.

Emad Aghajani
June 2020

Contents

Contents v

List of Figures ix

List of Tables xi

I Prologue 1

1 Introduction 5
1.1 Our Thesis 7
1.2 Contributions 7
1.2.1 List of Our Accomplishments 7
1.3 Outline 10

2 State of the Art 11
2.1 Empirical Studies on Software Documentation 13
2.1.1 Software Documentation Importance and Usage 13
2.1.2 Documentation Issues and Maintenance 15
2.1.3 API Documentation 16
2.1.4 Summing Up 17
2.2 Automating Developer Documentation 18
2.2.1 Summarization 18
2.2.2 Mining Crowd Knowledge 21
2.2.3 Summing Up 23
2.3 Summary and Conclusion 23

II Software Documentation: Automation and Challenges 25

3 A Large-scale Empirical Study on Linguistic Anti-patterns Affecting APIs 29
3.1 Motivation 31
3.2 Source Code Linguistic Antipatterns 32
3.3 Study Design 33
3.3.1 Research Questions 33
3.3.2 Context Selection 33
3.3.3 Data Extraction 34
3.3.4 Data Analysis 36
3.4 Results & Discussion 37
3.4.1 Impact of LAs on the Likelihood of Introducing Bugs 39
3.4.2 APIs Affected by LAs and Stack Overflow 41
3.5 Threats to Validity 43
3.6 Summary and Conclusion 44

4 Automated Documentation of Android Apps **47**
4.1 Introduction . . . 49
4.2 ADANA . . . 49
4.2.1 Building the ADANA Knowledge Base . . . 50
4.2.2 The ADANA Web Service . . . 55
4.3 Study Design . . . 58
4.3.1 Context Selection and Data Analysis . . . 59
4.3.2 Experimental Setting . . . 63
4.4 Results & Discussion . . . 65
4.4.1 ADANA Code Coverage . . . 65
4.4.2 ASIA Clone Detection Accuracy . . . 66
4.4.3 Impact of ADANA on Developers' Code Comprehension . . . 68
4.5 Threats to Validity . . . 71
4.6 Summary and Conclusion . . . 71

5 Software Documentation Issues Unveiled **73**
5.1 Introduction . . . 75
5.2 Study Design . . . 75
5.2.1 Data Collection . . . 75
5.2.2 Data Analysis . . . 78
5.3 Results & Discussion . . . 78
5.3.1 Information Content (What) . . . 80
5.3.2 Information Content (How) . . . 82
5.3.3 Tool Related . . . 85
5.3.4 Process Related . . . 87
5.4 Threats to Validity . . . 90
5.5 Summary and Conclusion . . . 91

6 Software Documentation: The Practitioners' Perspective **93**
6.1 Motivation . . . 95
6.2 Study Design . . . 96
6.2.1 Research Questions . . . 96
6.2.2 Context Selection: Surveys & Participants . . . 96
6.2.3 Data: Analysis . . . 100
6.3 What documentation issues are relevant to practitioners? . . . 100
6.3.1 Information Content (What) . . . 102
6.3.2 Information Content (How) . . . 104
6.3.3 Process/Tool Related . . . 106
6.4 What types of documentation are useful to practitioners? . . . 107
6.5 Threats to Validity . . . 109
6.6 Summary and Conclusion . . . 110

III Epilogue **111**

7 Conclusions and Future Work **115**
7.1 Contributions . . . 117
7.1.1 Empirical studies . . . 117
7.1.2 Supporting tools & frameworks . . . 118

7.2 Future Work 118
7.2.1 Short-term future work 119
7.2.2 Our long-term vision 120
7.3 Closing Words 122

Bibliography **123**

Artifacts' References **141**

IV Appendices **147**

A Code Time Machine **149**
A.1 Introduction 151
A.2 The Code Time Machine in a Nutshell 152
A.2.1 History Exploration 154
A.2.2 Time Traveling 155
A.3 The Code Time Machine In Action 155
A.4 Related Work 156
A.5 Conclusion and Future Work 156

Figures

3.1 An example question related to a method with a LA 43

4.1 The ADANA architecture 50
4.2 ADANA GUI 58
4.3 Tuning of the α, β, and t ASIA parameters 64
4.4 Commented code coverage (left) and average number of retrieved clones (right) . . . 65
4.5 Participants' understandability comparison with/without ADANA 69

5.1 Documentation Issues Taxonomy 79
5.2 Documentation Issues related to information content (what) 80
5.3 Documentation Issues related to information content (how) 83
5.4 Documentation Issues related to tools 86
5.5 Documentation Issues related to processes 88

6.1 Design of the two surveys used in our study 97
6.2 Documentation issues according to the results of Survey-I 101
6.3 Types of documentation perceived as useful by practitioners 107

A.1 *Code Time Machine* main window. 152
A.2 Displaying commit information by hovering over one 153
A.3 Different levels of detail in *timeline view* 153
A.4 Updating the *commit list view* by specifying active range 153
A.5 The *Diff window* 154
A.6 Example of *commits stack view* customizability and colorful mode 155

Tables

2.1 A summary of main types of contributions 13
2.2 Summary of some of the most notable previous studies on software documentation . 14

3.1 Linguistic antipatterns related to methods 32
3.2 Maven libraries and client projects considered 34
3.3 Number of libraries/releases/methods affected by LAs 38
3.4 Top-ten releases in terms of percentage of public API methods used by their clients . 39
3.5 Odds ratio by type of LA (significant results only) 39
3.6 Odds ratio of methods discussed in *Stack Overflow* 41
3.7 Linguistic Antipatterns found in methods with related questions on SO 42
3.8 Libraries having methods affected by LAs and discussed on SO 42

4.1 The heuristic to identify Java code snippets. 51
4.2 Original & standardized descriptions examples 52
4.3 Regular expressions used by description quality checker for cleaning descriptions . . 53
4.4 Regular expression used by the ones in Table 4.3 54
4.5 The 16 apps selected for RQ_1 59
4.6 Demographic of study participants (RQ_2) 61
4.7 Demographic of participants (RQ_3) 61

5.1 Study Dataset 76

6.1 Participants roles & programming experience 100

Part I

Prologue

Software documentation, the ideal companion of any software system, is intended to provide stakeholders with useful knowledge about the system and related processes. Despite its undeniable practical benefits during software development and evolution activities, however, its creation and maintenance have been often neglected, leading to inadequate and even inexistent documentation. In this thesis, we will study software documentation with the goal of automating its generation. Furthermore, we present techniques to help developers with code comprehension when documentation is missing, and present studies to understand the impact of documentation of developers and software systems.

Particularly in ***this part****, we discuss software documentation, its issues, previous work in this context, and roadblocks for creation of automated documentation generation systems. For that, in* ***Chapter 1****, we briefly review the state of the art highlighting the most notable studies in the context of this thesis. Then we present our thesis that* empirical knowledge about software documentation issues experienced and considered relevant by practitioners is instrumental to lay the foundations for the next-generation tools and techniques for automated software documentation. *Finally, we provide the list of our accomplishments.*

Afterwards, in ***Chapter 2****, we review previous studies concerning developer documentation and summarize their main contribution, as well as their limitations. We also briefly mention how we will address some of these issues with our studies.*

1

Introduction

A "software post-development issue" [fCMA12]. An after-thought, so to speak. This is how the ACM Computing Classification Systems (CCS)[1] categorizes **software documentation**. Although peculiar, this classification aligns well with the general perception that there are more exciting things to do than documenting software, especially if said software has already been developed.

Good documentation, the ideal companion of any software system, is intended to provide stakeholders with useful knowledge about the system and related processes. Depending on the target audience, the contents of documentation varies. For example, user documentation (*e.g.,* user manuals) explains to end-users how they should use the software application, while technical documentation (*e.g.,* API reference guides) describes information about the design, code, interfaces and functionality of software to support developers in their tasks.

Despite the undeniable practical benefits of documentation during software development and evolution activities [FL02, LSF03b, KM05, ZGYS+15], its creation and maintenance have been often neglected [FL02, FWG07, CH09, LVLVP15, KM05, ZGYS+15], leading to inadequate and even inexistent documentation. Many studies reported software documentation as being affected by insufficient and inadequate content [Rob09, RD11, UR15], obsolete and ambiguous information [UR15, WNBL19], and incorrect and unexplained examples [UR15], to name just a few issues. In contrast to this rather sad *status quo*, not only are there studies that attest that documentation is actually useful [FL02, CH09, DR10, RD11, GGYR+15], but also it simply makes sense to document software—it is just not an activity enjoyed by many.

To address these issues (at least partially), different approaches and tools have been proposed to aid developers during software documentation, including **automated summarization approaches** [MAS+13, YR13, FCR+17, CCLVAP14, YR14, MM16, LVLVP16, LVLV+16], by creating extractive or abstractive summaries [HAMM10]. While in the former a subset of code/comment elements is selected from the code chunk to describe it, the latter includes information which is not explicit in the original document [HAMM10]. However, if the information to comprehend the code is simply not there, these approaches fall short.

Fulfilling such requirements/needs has been the goal of several works aimed at **automatically documenting software artifacts** [FL02, MM16]. As a result, a variety of different automated approaches for the generation and recommendation of documentation (*e.g.,* [MAS+13, PBDP+14, MM16, RJAM17, HLX+18]) have emerged. Such "recommendation systems" have been designed with the goal of retrieving and suggesting relevant pieces of information (*e.g.,* documentation, Stack Overflow discussions [Sta17]) for a given piece of code inspected in the Integrated Development

[1] See `https://dl.acm.org/ccs`

Environments (IDE).

Although these techniques have demonstrated their ability in producing useful documentation, they still exhibit a number of major limitations. When generating recommendations, for instance, most of them tend to work at a coarse-grained level, not allowing to document a single line of code of interest (*i.e.,* to explain what is implemented by that line). Also, these recommenders generally focus on one specific source of information (*e.g.,* Stack Overflow discussions), not taking advantage of the heterogeneous nature of the information that can be found in the official documentation, in the project-related repositories (*e.g.,* issue tracker), and in general online resources.

Moreover, the source code on which a developer is working is the primary input to most of these approaches, regardless of other parameters that vary from person to person or task to task, and can be taken into account for more relevant and useful results. On this point, Binkley *et al.* [BLH+13] stress the importance of "useful" documentation, besides mere "good" documentation, and emphasize the necessity of exploiting non-code factors, such as the expertise level of the developer who the information is being given to. Moreover, Happel *et al.* [HM08] conducted a survey on some well-known recommender systems and discussed their limitations and extant challenges, indicating that obtaining useful documentation is not a straightforward task and there are several challenges to be addressed.

In a recent discussion on the future of software artifact documentation by Robillard *et al.* [RMT+17], they suggest a paradigm shift towards systems that automatically generate documentation in response to a developer's query, while considering her working context. In this proposal, they conducted a review of state-of-the-art approaches and outlined the key challenges in three categories: information inference (*i.e.,* mechanisms to model and infer information), document request (*i.e.,* mechanisms to enable developers to express their information needs in a better way) and document generation (*i.e.,* approaches to generate appropriate output).

While the creation of such novel systems entails conceptual and technical challenges related to the collection, inference, interpretation, selection, and presentation of useful information, it also requires solid empirical foundations on software developers' needs — *what* information is (or is not) useful *when* to developers. Previous studies have investigated different aspects of documentation (*e.g.,* needs, learning obstacles) with the general goal of identifying the root causes of documentation issues (*e.g.,* inaccuracy, outdatedness). Although existing studies have revealed some of these needs through interviews with and surveys of practitioners, their results are limited by the low number [FL02] and lack of diversity [Rob09] of practitioners questioned and documentation artifacts analyzed.

1.1 Our Thesis

We formulate our thesis as follows:

> *"Empirical knowledge about documentation issues experienced and considered relevant by practitioners is instrumental to lay the foundations for the next-generation tools and techniques for automated software documentation."*
>
> Emad Aghajani, 2020

As mentioned, the creation of future automated documentation tools requires solid empirical foundations on software developers' needs. We ran into the same roadblock in our preliminary studies. For instance, while developing a framework to generate code comments for a given piece of code, we learnt that we are not able to tackle many issue due to the lack of empirical knowledge in this research area.

To make up for this missing empirical knowledge, we conducted three large-scale studies focusing on a different aspects of software documentation such as:

- Impact of poor documentation of software quality
- Software documentation issue types faced by documentation users and developers
- The relevance of different types of documentation issues to software practitioners
- The usefulness of different types of software documentation in context of different software activities

All of this can be leveraged to revise current understanding about the nature of software documentation, and revamp the current approaches for automated software documentation, thus laying the foundations for a novel generation of recommender systems in this field.

1.2 Contributions

The contributions of our research can be grouped in two high-level categories:

Empirical studies on software documentation. On the empirical side, we conducted several studies and surveys to better understand the nature of software documentation, as well as practitioners' needs in this context. We also investigated software documentation issues and their impact on software systems. To denote the type of achievements we use 📘 icon for publications, 💬 icon for discussions, and 🖧 icon for catalogs/taxonomies.

Supporting tools and frameworks. On the practical side, we devised techniques and approaches to support developers with code comprehension, where ADANA, a novel approach for automating developer documentation, is our most notable accomplishment. To denote the type of achievements we use ⚙ icon for tools (*e.g.,* an IDE plugin).

1.2.1 List of Our Accomplishments

In the following we enlist our main contributions and accomplishments. A similar list of accomplishments can also be found in the beginning of each chapter.

- ► **The Impact of Poorly Documented APIs** → Chapter 3
Given the importance of *comprehensibility* and *usability* in the context of APIs, we conjectured that poorly documented APIs can be problematic for client projects using them. To validate our conjecture, we performed a large-scale study to investigate (i) the likelihood of introducing more bugs in the client projects using such APIs, and (ii) whether developers tend to ask more questions on *Stack Overflow* about such APIs.

> **A Large-scale Empirical Study on Linguistic Antipatterns Affecting APIs** [ANBL18]
> Emad Aghajani, Csaba Nagy, Gabriele Bavota, Michele Lanza
> In *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*, pp. 25–35. IEEE, 2018.

- ► **Automated Generation of Documentation (**ADANA**)** → Chapter 4
To partially address the lack of good documentation, we developed ADANA, a framework which generates fine-grained code comments for a given piece of code at the granularity level intended by the developer.

> **Automated Documentation of Android Apps** [ABLVL19]
> Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza
> In *IEEE Transactions on Software Engineering*, To be published

- ► ADANA **Android Studio plugin** → Chapter 4
We implemented the ADANA approach in form of an IDE plugin. Using our Android Studio plugin, a developer can select a snippet of code she is interested in comprehending, and then invokes ADANA. If our approach succeed, a code comment retrieved by ADANA explaining the selected code snippet will be injected into the code. Our plugin is publicly available.

- ► ASIA **Clone Detector** → Chapter 4
The ADANA approach relies on code clone detection. For that, we devised ASIA, an approach built on top of standard IR clone detection and tailored for identifying clones in Android-related code.

- ► **Studying Software Documentation Issue Types** → Chapter 5
We present a systematic study on software documentation issues. We qualitatively analyzed different types of artifacts from diverse data sources and identified the *issues that developers face when dealing with documentation*.

> **Software Documentation Issues Unveiled** [ANVM+19]
> Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza
> In *Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*, pp. 1199–1210. IEEE, 2019.

- ► **Taxonomy of Documentation Issue Types** → Figure 5.1
Based on our analysis on software documentation issues [ANVM+19], we built a *comprehensive taxonomy consisting of 162 types of documentation issues* faced by developers and users of software documentation, linked to (i) the information it contains, (ii) how the information is presented, (iii) the documentation process and (iv) documentation tool support.

- ► **In-Depth Discussion of Documentation Issues and Implications** → Chapter 5
As part of our qualitative analysis on software documentation issues [ANVM+19], we carefully discuss each category of issues, and present interesting examples and common solutions. We

also discuss their implications in software research and practice, deriving a series of actionable proposals needed to address them both for researchers and practitioners.

- **Artifact Labeling Web app** → Chapter 5
  To support our manual investigation of software artifacts, we built a Web app which enables us to label online artifacts at sub-sentence level and to resolve conflict among taggers.

- **Practitioners' Perspective on Software Documentation** → Chapter 6
  To better understand the relevance of our previous findings to practitioners, we performed two surveys focusing on the documentation issues that practitioners perceive as more relevant, together with the solutions they apply when these issues arise, and the types of documentation that practitioners consider important given specific tasks.

  > **Software Documentation: The Practitioners' Perspective** [ANLV+20]
  > Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, David C. Shepherd
  > In *Proceedings of 42nd ACM/IEEE International Conference on Software Engineering (ICSE 2020)*, To be published. IEEE, 2020.

- **Our Vision on the Future of Automated Software Documentation** → Section 7.2
  To achieve our goal of high-quality automatic documentation generation, we present our vision on the future of automated software documentation, with a focus on context-awareness. In our short paper we review the state of the art and highlight their current most notable limitations. Finally, we picture our ideal recommender system and discuss requirements and obstacles to materialize it.

  > **Context-Aware Software Documentation** [Agh18]
  > Emad Aghajani
  > In *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*, Doctoral Symposium, pp. 727–731. IEEE, 2018.

- **Code Time Machine** → Appendix A
  We provide *Code Time Machine*, a lightweight IDE plugin which uses visualization techniques to depict the history of any chosen file augmented with information mined from the underlying versioning system.

  > **The Code Time Machine** [AMBL17]
  > Emad Aghajani, Andrea Mocci, Gabriele Bavota, Michele Lanza
  > In *Proceedings of 25th IEEE International Conference on Program Comprehension (ICPC 2017)*, pp. 356–359. IEEE, 2017.

1.3 Outline

This dissertation is composed of seven chapters and one appendix, organized into three parts as follows:

Part I: Prologue groups introductory chapters including this introduction.

- **Chapter 2** presents and discusses the state of the art. The chapter addresses different topics concerning this dissertation, including software documentation, its impact on software systems, its issues and potential solutions.

Part II: Software Documentation: Automation and Challenges is devoted to our main accomplishments with respect to software documentation issues.

- **Chapter 3** introduces our initial study on the effect of poor documentation of software systems.
- **Chapter 4** addresses the missing documentation issue by introducing ADANA, a novel approach for automating documentation
- **Chapter 5** presents our large-scale study which led to a detailed taxonomy of 162 types of issues faced by developers and users of software documentation.
- **Chapter 6** presents our follow-up study on documentation issues, where we perform two surveys with practitioners to learn more about the significance of documentation issues. Additionally, we study the types of information which are more important to developers in different contexts.

Part III: Epilogue concludes this dissertation with the final chapter of this thesis.

- **Chapter 7** presents conclusion remarks and directions for future work.

Part IV: Appendices present our complementary studies and tools in appendices.

- **Appendix A** presents our tool-demo paper about *Code Time Machine*, a lightweight visualization tool to support developers with code evolution comprehension.

2

State of the Art

THIS CHAPTER reviews previous studies concerning developer documentation, *i.e.,* any type of artifacts exploited by developers in software development and maintenance activities. A growing body of literature have been focused on software documentation, and researchers have investigated software documentation from different perspectives. In this context, the two major lines of research related to software documentation can be classified as following:

1. empirical studies investigating different documentation aspects (*e.g.,* quality)

2. tools and approaches to automatically generate or recommend documentation

Correspondingly, we summarize relevant studies in Sections 2.1 and 2.2. Each of the two sections is concluded with a short discussion of previous work limitations.

Structure of the Chapter

- **Section 2.1** summarizes (empirical) studies on the nature of software documentation aimed at investigating different aspects such as documentation issues or developer concerns.
- **Section 2.2** focuses on tools and approaches built to help developers with code comprehension. In particular, we are interested in studies that suggest or generate documentation (*e.g.,* by automatically generating code comments), on one hand, and those summarizing source code, on the other hand.
- Finally, **Section 2.3** concludes this chapter by discussing the state of the art limitations.

2.1 Empirical Studies on Software Documentation

Developers have different views on how code should be documented and what type of documentation is useful. For instance, code comments outlining an algorithm or elaborating a data structure are usually perceived as important, and missing code comments is favored over useless or misleading code comments [Spi10]. Therefore, to achieve high-quality documentation, we require first a deep understanding of documentation characteristics and developers' needs.

A variety of empirical studies have targeted software documentation artifacts with different aims. Table 2.1 categorizes some of the previous studies based on the type of contribution they made. Moreover, Table 2.2 provides a summary of some of the most related studies.

Table 2.1. A summary of main types of contributions

Type of Contribution	Studies
Reporting evidence of documentation importance and its usage in specific phases of the software lifecycle	[FL02, CH09, DR10, KM05, RD11, GGYR+15, dSAdO05, GGM+13]
Describing problems that developers face when dealing with documentation	[KM05, CH09, Rob09, RD11, UR15]
Listing quality attributes required in documentation, *e.g., clarity, completeness*	[AS92, Dau11, RD11, GGYR+15, UR15, PDS14]
Providing recommendations for constructing documentation, *e.g., standards*	[FL02, LSF03a, VC04, KM05, Rob09, DR10, RD11, GGYR+15, UR15]
Proposing frameworks and tools for evaluating documentation concerns, *e.g., cost, benefit and quality attributes*	[AS92, Dau11, GGYR+15, AAHMA16, SMAR17]

The rest of this section reviews such studies aimed at investigating different aspects of software documentation in three steps.

- **Section 2.1.1** focuses on studies discussing the role of documentation and its different types in software development lifecycle periods, *e.g.,* maintenance.
- **Section 2.1.2** reviews studies related to software documentation issues.
- **Section 2.1.3** focuses on studies specifically related to API reference documentation, as it builds the foundation needed for Chapter 3, where we particularly study this type of documentation.

2.1.1 Software Documentation Importance and Usage

Software documentation comes in a variety of types, each specialized for particular purposes. To learn more about how different documentation types are employed by developers, some studies have investigated software documentation usage. As a result, these studies have revealed how documentation is used during certain software development lifecycle periods, *e.g.,* maintenance. For instance, Kajko-Mattsson [KM05] carried out an exploratory study with 18 Swedish organizations and found that documentation is usually neglected during corrective maintenance.

Garousi *et al.* [GGM+13, GGYR+15] conducted an industry case study to evaluate the impact of attributes such as documentation type, developer's role, degree of experience on documentation usage and usefulness of a number of documentation types (using the taxonomy by Zhi *et al.*

Table 2.2. Summary of some of the most notable previous studies on software documentation

Study	**Artifacts**	**Summary of findings** (related to concerns and quality attributes)
Forward and Lethbridge (2002) [FL02]: Questionnaire with 48 participants (from sw industry, research peers, and mail lists members).	Software documentation regularly used by participants	Despite documentation being outdated, practitioners learn how to deal with it. *"Software documentation tools should seek to better extract knowledge from core resources. Software professionals value technologies that improve automation of the documentation process, and its maintenance."*
Kajko-Mattsson (2005) [KM05]: Exploratory study with 18 Swedish organizations.	Maintenance-related documentation artifacts	*"Documentation within corrective maintenance is still a very neglected issue."*
Chen and Huang (2009) [CH09]: Questionnaire with 137 project managers and sw engineers of the Chinese Information Service Industry Association of Taiwan.	Software documentation regularly used by participants	Most typical problems in software documentation quality for maintenance are that software documentation is *untrustworthy, inadequate, incomplete or does not even exist, lacks traceability, does not include its changes, and lacks integrity and consistency*.
Robillard (2009) [Rob09]: Personal interviews with 80 professionals at Microsoft.	API documentation and source code	The top obstacles for API learning are: *resources for learning* (documentation, examples, *etc.*), *API structure, Background, Technical environment, Process.* API documentation must *include good examples, be complete, support complex usage scenarios, be organized, and have better design*.
Dagenais and Robillard (2010) [DR10]: (i) A qualitative study with 12 contributors and 10 users of open-source projects, and (ii) an evolution analysis of 19 documents from 10 open-source projects.	Open-source projects documentation in a repository or wiki (*e.g.,* Django, Firefox and Eclipse)	In open-source projects, knowing the relationships between documentation and decisions of contributors help to define better techniques for documentation creation and maintenance. When a wiki is selected to host documentation, its quality is threatened by erroneous updates, SPAM or irrelevant content (URLs included). This requires more effort for maintaining wikis.
Robillard and Deline (2011) [RD11]: (i) An initial questionnaire, (ii) a set of qualitative in-person interviews and (iii) a questionnaire with 440 developers at Microsoft.	API documentation regularly used by participants	Relevant issues in the documentation that affect the developers learning experience: *"documentation of intent, code examples, cookbooks for mapping usage scenarios to API elements, penetrability of the API, and format and presentation of the documentation."*
Plösch et al. (2014)[PDS14]: Online questionnaire with 88 software professionals, mainly German speakers.	Software documentation regularly used by participants	The most important attributes are accuracy, clarity, consistency, readability, structuredness and understandability. *"There is a need for automatic analysis of software documentation quality."*
Zhi et al. (2015) [ZGYS+15]: Mapping study about a set of 69 papers from 1971 to 2011.	N/A	Documentation quality attributes that appear in most of the papers are *completeness, consistency and accessibility*. More empirical evidence is required involving large-scale development projects, or larger samples of participants from various organizations; more industry-academia collaborations are also required, and more estimation models or methods to assess documentation.
Garousi et al. (2015) [GGYR+15]: Industry case study with analysis of documentation (using the taxonomy by Zhi *et al.* [ZGYS+15]) and a questionnaire with 25 employees of NovAtel Inc.	Source code and a sample of software documentation (design, tests and processes)	Technical documentation is preferred during development than during maintenance tasks; the preferred source of information for maintenance is the source code; other sources of information have no significant impact on developers' preferences.
Uddin and Robillard (2015) [UR15]: A case study and a questionnaire with 230 software professionals from IBM.	API documentation	The top 10 problems in API documentation are (i) incompleteness, (ii) ambiguity, (iii) unexplained examples, (iv) obsoleteness, (v) inconsistency and (vi) incorrectness; while in presentation are (vii) bloat, (viii) fragmentation, (ix) excess structural information and (x) tangled information.
Alhindawi et al. (2016) [AAHMA16]: Topic-modeling-based study.	KDE/KOffice source base and its external documentation	A novel approach for evaluating documentation quality. Tools that can automatically assess the software documentation quality are highly demanded. Labeling and grouping documentation would impact its quality positively.
Sohan et al. (2017) [SMAR17]: Controlled study with 26 software engineers.	WordPress REST API documentation	Developers feel more satisfied when having examples. When documentation lacks examples, developers spend more time on coding, execute more trial attempts, and have lower success rates.

[ZGYS+15]). This study has revealed that though technical documentation is the preferred source of information during development, developers prefer source code to documentation for maintenance activities. They also observed developers tend to use design documents during the development phase, while code comments are considered the most useful documentation artifacts for maintenance purposes. Additionally, they also studied the impact of 10 quality attributes (*e.g.,* readability, accuracy, and completeness) on the overall perceived quality of documentation. They found readability, the relevance of content, and organization are quality attributes which have the most significant impact on the overall perceived quality of documentation.

To further investigate the types of documentation artifacts that are the most important during the maintenance phase, De Souza *et al.* [dSAdO05] conducted a survey of 76 software maintainers, mostly from Brazil. The findings show that source code, code comments, data model, and requirement specification are at the top of the list. This finding is in line with the results of Garousi *et al.* [GGM+13, GGYR+15].

Some other works focus their attention on developers' opinions about software documentation. In a survey with 48 participants from software industry, conducted by Forward and Lethbridge [FL02], they found that practitioners learn how to deal with documentation despite documentation being outdated. The study showed that participants value software documentation tools that extract knowledge from core resources, including the system's source code, test code, and changes to both. They concluded that professionals value technologies that improve automation of the documentation process and its maintenance.

More recently, Sohan *et al.* [SMAR17] performed a controlled study with 26 software engineers, focusing on *WordPress REST API* documentation. Their findings highlight the importance of examples in the documentation. They found that developers spend more time on coding, execute more trial attempts, and have lower success rates when documentation lacks examples.

Open-source projects documentation has been studied by Dagenais and Robillard [DR10]. In particular, they performed (i) a qualitative study with 12 contributors and 10 users of open-source projects, and (ii) an evolution analysis of 19 documents from 10 open-source projects. Their findings show that better techniques for documentation creation and maintenance could be adopted if the relationships between documentation and decisions of contributors are known. They also found that when a wiki is selected to host documentation, its quality is threatened by erroneous updates, SPAM or irrelevant content (URLs included), hence requiring more effort for maintaining wikis.

Moreover, Alhindawi *et al.* [AAHMA16] investigated *KDE/KOffice* codebase and its external documentation with the goal of studying the documentation quality. They proposed a novel approach for evaluating documentation quality and developed tools that can automatically assess the software documentation quality. They also argue that labeling and grouping documentation would be very efficient for the developer's progression and would positively impact documentation quality.

2.1.2 Documentation Issues and Maintenance

There are also studies focused on developers' issues when dealing with documentation. For instance, Chen and Huang [CH09] surveyed 137 project managers and software engineers of the Chinese Information Service Industry Association of Taiwan. Their results suggest that *lack of traceability, untrustworthiness, and incompleteness or non-existence* are among the most important documentation issues for maintainers.

Robillard and Deline [RD11] investigated API learning obstacles faced by developers. For that, they conducted two questionnaires with 440 developers at Microsoft, followed by a set of qualitative in-person interviews. Their findings stress the importance of some documentation components such as code example, developers' intent, and mapping from usage scenarios to API elements, as well as

its format and presentation. In general, their findings also emphasized the importance of reference documentation when learning how to use an API and motivated several studies to understand the essential elements needed to properly document APIs.

In a very similar study, Robillard [Rob09] reported on the results of a survey conducted with 83 professionals at Microsoft concerning API documentation and source code. In particular, the author interviewed subjects of the study with respect to obstacles they face when learning new APIs. His findings highlight the top obstacles for API learning, such as the *lack of good examples* or *complex usage scenarios in the documentation. He also notes that the rapid growth of APIs increases the need for the tools supporting developers to identify the information they need.*.

Similarly, Watson *et al.* [WSJSS13] reviewed the API documentation of 33 popular libraries to verify whether it includes the elements of desirable API documentation defined in previous work. They found that most of the analyzed documentations included most (or all) the aspects of desirable API documentation, with a high standard for writing quality.

Plösch *et al.* [PDS14] conducted an online questionnaire with 88 software professionals, mainly German speakers. Their findings revealed that *accuracy, clarity, consistency, readability, structuredness, and understandability* are among the most important documentation attributes. They also found that automatic analysis of software documentation quality needs more attention from researchers and practitioners. Finally, their data suggests that documentation standards (*e.g.,* IEEE Std.1063-2001, ISO 26514:2008) are not considered to be important by developers.

Furthermore, Uddin and Robillard [UR15] carried out a case study and a questionnaire with 230 software professionals from IBM with the goal of studying the developers' issues with the API documentation. Their finding unveiled ten common problems with API documentation, namely: *incompleteness, ambiguity, unexplained examples, obsoleteness, inconsistency*, and *incorrectness*; while in the presentation are *bloat, fragmentation, excess structural information*, and *tangled information*.

Finally, a number of studies leveraged a mining-based strategy for identifying documentation issues discussed by developers [LVDP13, BTH14, RS16] or by application users [KSNH15]. In this regard, we present an extensive taxonomy of 162 types of issues faced by developers and users of software documentation. For that, we conducted a manual analysis of 878 documentation-related artifacts from four different sources (*e.g.,* Stack Overflow discussions, pull requests, developers' mailing lists) [ANVM+19]. The taxonomy is followed by a discussion of implications for developers and researchers which exposes good practices and interesting research avenues in software documentation (see Chapter 5).

2.1.3 API Documentation

Among different types of documentation, API reference documentation holds a special place among other types of documentation, viewed among the most useful sources of information by developers [RD11]. Therefore, another portion of documentation-related literature focuses on this specific type of documentation by (i) investigating APIs usability, design, and documentation; and (ii) techniques and tools to help developers in using APIs.

Several studies focused the attention on the API usability and factors promoting/hindering it. McLellan *et al.* [MRTS98] suggest the need for usability tests for APIs in the same way in which usability tests are performed in the context of user interface design. Myers and Stylos [MS16] echo such a recommendation, indicating usability as one of the key factors to optimize when designing an API, no less important than its correctness.

Ko *et al.* [KMA04] showed the difficulties experienced by developers when dealing with APIs requiring the use of multiple objects. Stylos and Myers [SM08], inspired by this finding, ran a user study to investigate the role played by method placement (*i.e.,* which class the method belongs to) in

the usability of APIs requiring the use of multiple objects. Their findings show that method placement plays an important role, strongly impacting developers' performance when dealing with APIs.

Ellis *et al.* [ESM07] ran a user study to assess the impact on the API usability of the factory design pattern as compared to the adoption of simple class constructors. They observed that, in many situations, adopting the factory pattern significantly lowers the API usability.

Stylos and Clarke [SC07] investigated whether programmers are more effective when using APIs requiring constructor parameters as compared to parameterless default constructors. Their findings highlight the strong preference (and higher effectiveness) programmers have for APIs that do not require constructor parameters.

Piccioni *et al.* [PFM13] performed a study with 25 programmers to investigate API usability. The study takes advantage of a combination of interviews with the participants and systematic observation of their behavior during programming tasks. Among the findings they report, they highlight the difficulty of defining proper names when designing an API. This is confirmed by Arnaudova *et al.* [ADPA16] who present a catalog of 17 linguistic antipatterns in source code, capturing inconsistencies among the naming, documentation, and implementation of attributes and methods.

Duala-Ekoko and Robillard [DER12] conducted a controlled experiment with 20 developers to understand the types of questions they ask when facing unfamiliar APIs. Overall, the authors collected over 20 hours of screen-captured videos spanning 40 implementation tasks. As part of their findings, the authors report that developers have difficulties guessing an API semantic from its name.

Maalej and Robillard [MR13] proposed a taxonomy of knowledge types in API reference documentation by investigating the documentation of two popular frameworks. Their taxonomy overviews the types of information reported in APIs documentation and can be used by developers to evaluate the content of their documentation.

Finally, Acar *et al.* [ABF+17] studied whether the usability of APIs provided by several cryptographic libraries impacts the ability of developers to create secure code. Their study has been conducted with 256 Python developers and shows that APIs designed for simplicity (*e.g.,* guiding the developers by reducing the decision space) are not always enough since poor documentation or the lack of code examples can still hinder developers' ability to cope with them. On the other side, good documentation and examples can make developers comfortable to work with complex APIs.

2.1.4 Summing Up

On top of these individual studies, the mapping study by Zhi *et al.* [ZGYS+15] is notable as it reviews 69 documentation-related papers from 1971 to 2011. Authors conclude that some documentation aspects such as documentation quality, benefits, and cost are neglected and more estimation models or methods are needed. They found *completeness, consistency, and accessibility* to be the most frequently discussed documentation quality attributes in the existing literature. Furthermore, they also call for more and stronger empirical evidence, larger-scale empirical studies, and more industry-academia collaborations. Some of this thesis's contributions (*i.e.,* Chapter 5 and Chapter 6) go exactly in this direction.

Most of the aforementioned studies gathered information directly from participants and used samples restricted to a specific context (*e.g.,* a company). These studies are therefore not diverse enough in terms of analyzed artifacts and programming languages used by developers, and the largest samples reported in the studies are 440 (Robillard and Deline [RD11]) and 230 practitioners (Uddin and Robillard [UR15]).

To overcome the limitations imposed by interviews and surveys, we conducted a large-scale qualitative study [ANVM+19] to identify the issues that developers face when dealing with documentation. For that, we opted for an approach that allowed us to study a wider population in terms of

number and types of artifacts by mining different data sources. Our results complement previous categorizations of documentation issues with a taxonomy that considers documentation content, processes and tools. Moreover, it is the first mining-based study focused on identifying documentation issues as discussed by practitioners in software repositories. Previous studies following a mining-based strategy are more general, identifying topics discussed by developers [LVDP13, BTH14, RS16] or by apps' users [KSNH15]. We discuss the study details in Chapter 5.

Although previous studies such as [Rob09, RD11, UR15, GGYR+15, ZGYS+15, dSAdO05] have investigated documentation issues and documentation types, there is still a gap when determining which are the more relevant ones — both issues and documentation types — to practitioners. Some of previous studies focused on the issues experienced with specific types of documentation [UR15, Rob09, RD11], or only during a certain development phase [CH09, KM05, dSAdO05]; and while there are studies reporting the usefulness of certain types of documentation based on studies with developers [GGYR+15, ZGYS+15, dSAdO05], those studies involved a small sample of participants [GGYR+15], did not take practitioners' perspective into account [ZGYS+15], covered few documentation types [GGYR+15] or quality attributes [ZGYS+15], or focused on listing documentation preferences of developers during a certain development phase [dSAdO05, GGM+13, GGYR+15] and did not provide rationale for the preferences [dSAdO05].

We address those limitations by analyzing what are the most relevant documentation issues and the most useful documentation types for developers, while trying to be comprehensive in terms of the issues, documentation types, and developers activities [ANLV+20]. We not only report the issues and documentation types, but also the developers' reasons for their choices. Besides, we also present the common solutions that practitioners adopt when dealing with each documentation issue. This study is further elaborated in Chapter 6.

2.2 Automating Developer Documentation

Supporting developers in the code comprehension process with the aim of automatically documenting software has been extensively studied by researchers. Previous automatic documentation approaches rely on summarization techniques (*e.g.,* [HAMM10, RMM+14b]), and mining of repositories with developers' communications and Q&A websites and forums (*e.g.,* [RRK15, VPDPC14]).

In the following, we review the state of the art studies on software artifacts summarizations (2.2.1), followed by ones aiming at automated suggestion or generation of documentation (2.2.2).

2.2.1 Summarization

Code summarization has become one of the mainstream methods for the automatic documentation of source code. Concise natural language descriptions of source code fragments enhance the code comprehension process by reducing the amount of code to be read by the developer and, as a result, the time required to understand the code. The proposed summarization techniques fall into two categories: (1) key-words extraction [ERKC13, HAM10, RMM+14b, AT15]; and (2) natural language document generation [RMB11, SHM+10].

Natural language summarization can be done in two ways [NM+11]: extractive or abstractive. Extractive summaries [TR16, SPVS11b] are obtained by selecting a subset of document elements (*e.g.,* a subset of code statements) which represents the most important information in the code. Note that this is not really a piece of documentation that can help in comprehending complex code, but rather a way to save time to developers by removing "semantically-irrelevant" statements. Abstractive approaches [HAMM10, MM16, RMB11], on the other hand, take the semantics of the text and apply natural-language processing techniques to generate a summary.

Moreover, these approaches produce summaries with different levels of granularity and are designed to summarize different types of documents. In the following, we review some of the most notable studies in each category.

Summarizing methods Summarizing methods and its inner elements (*e.g.,* parameters) has been vastly studied by researchers. In a seminal work, Haiduc *et al.* [HAMM10] carry out some experiments on use of automatic text summarization techniques such as Vector Space Model (VSM), Latent Semantic Indexing (LSI), to generate source code summaries. Their case study on two Java systems indicate that the combined summaries achieves a better result. They also found that summaries containing the full identifiers, as opposed to the ones where the identifiers are split are preferred by developers.

Rodeghero *et al.* [RMM+14b] improve Haiduc *et al.* accomplishments [HAMM10] by presenting an improved keyword summarization approach with the goal of summarizing Java methods. For that, they conducted an eye-tracking study about the statements and keywords that programmers consider as important when they summarize source code, highlighting the importance of the method signature and method invocations keywords in this regard. They apply this empirical findings to build an improved VSM-based summarization tool by modifying the weights assigned to different keywords. Their evaluation study indicated that their new approach outperforms the same-class state-of-the-art approach by Haiduc *et al.* [HAMM10].

Source code summarization techniques, however, are not limited to keywords generation. For instance, Sridhara *et al.* [SHM+10]) propose an automatic approach to generate natural language text describing the overall actions of a given Java method. For that, they designed heuristics to identify and choose key statements in the given method, and then used keywords from selected statements to generate and synthesize a natural language summary using sentence templates. The conducted evaluation study shows that in developers' opinion generated summaries are concise, accurate, and do not miss important content.

Sridhara *et al.* [SPVS11a, Sri12], follow-up to their previous study, present a novel approach to automatically identify code fragments implementing a high-level actions, and to describe them as a natural language summary. Their approach extracts and leverages different types of source code information, *e.g.,* abstract syntax tree and control flow graph, as well as several heuristic for identifying such coherent code statements. For summarization, they apply similar summarization techniques as used in the previous work [SHM+10].

McBurney *et al.* [MLMW14] present a source code summarization approach based on topic modeling. In their approach they consider each Java method as a document for creating a topic model. Given the list of topics in each method, they build a hierarchy of topics, with more general topics near the top of the hierarchy. The authors evaluation study shows that in majority of cases, the generated keywords (*i.e.,* topics) are at least "somewhat accurate".

Dragan *et al.* [DCM06] propose a taxonomy for object-oriented class method stereotypes by unifying the literature on method stereotypes. Based on this taxonomy, they defined a set of rules and developed a tool, named *StereoCode*, for reverse engineering method stereotypes in C++ language using static analysis techniques. Moreover, their tool can annotate class methods with stereotype information. However, the augmented information using this approach is limited to the stereotype category (*e.g.,* `/** @stereotype set */` or `/** @stereotype property */`).

Some studies also focused on specific aspects of methods, *e.g.,* method parameters [SPVS11b] or method usages [MM14, MM16]. For example, Sridhara *et al.* [SPVS11b] present an approach to summarize parameter usage in a given method. For that, the approach identifies the

code statements in which the parameter is used. Using a set of heuristics, the approach selects the statements which are estimated to be closer to the method's computational intent. Finally, the approach generates a natural language phrase for the selected statements. The evaluation study indicates that their approach improves developers' comprehension of the purpose of methods' parameters by providing useful and accurate summary.

Concerning summarization of method usages, McBurney and McMillan [MM14, MM16] developed an approach to generate a summary describing a given method's context, *i.e.,* how a method is used in its context. Their approach uses the PageRank algorithm [LM11] to identify the most important methods in the given method's context and generates different types of summaries using the keywords extracted based on the actions being performed by identified methods in the context.

Summarizing classes Although some of the previous studies do not make a distinction between method and class (*e.g.,* [HAMM10]), there are studies focused on summarizing classes. For example, Fowkes *et al.* [FCR+17] introduced a new technique for class summarization by automatically folding non-essential regions of a class. For that, they use a novel topic model for source code which identifies which tokens are most relevant in their context. The evaluation study demonstrates that this approach is favored by experienced developers and outperforms several baselines.

Inspired by their previous work at method level [DCM06], Dragan *et al.* [DCM10] present a taxonomy of class stereotypes derived from an empirical investigation of 21 open-source systems written in C++. In addition, they developed an approach to automatically determine the stereotype of a class and redocument it based on the frequency and distribution of method stereotypes introduced in their previous work.

Following the same idea, Moreno *et al.* [MAS+13, MMPVS13] presented a technique to automatically document Java classes based on stereotypes. In this approach, they consider methods with similar stereotypes to their parent class's, and then generate and combine natural language descriptions for selected methods using sentence templates.

Summarizing cross-cutting concerns Rastkar *et al.* [RMB11] introduce an approach which automatically generates a description for a crosscutting concern in the code, in the form of an abstractive summary. Given a set of methods implementing a crosscutting concern, the approach identifies patterns and key similarities using both structural and natural language information from the source code. Finally, the approach constructs a textual summary based on a set of predefined template sentences and extracted information.

Summarizing code snippets Ideally a source code summarization technique should be able to provide a summary for a given fragment of code at whatever granularity. This leads us to a set of studies focusing on automatic documentation of source code snippets. In this context, Ying *et al.* [YR13, YR14] explored the usage of machine learning for selecting lines in a code fragment that should be in an extractive summary. As opposed to that, our proposed approach [ABLVL19], ADANA, can document a snippet of code with descriptions mined from the Web.

Others There are also other studies designed to work with specific software artifacts. For instance, Li *et al.* [LVLVP18] present a catalog of 21 stereotypes for methods in unit tests, *e.g.,* `TestInitializer` and `NullVerifier`. Based on this catalog, they implemented an approach, named *TeStereo*, to automatically detect unit tests methods stereotypes, relying on static control-flow and dataflow. Their findings suggests that *TeStereo* is able to detect unit test stereotypes with low error, and proposed stereotypes help developers with tests cases comprehension.

Additionally, there are other summarization techniques focusing on other peripheral software artifacts such as bug reports [LMC12, LMC15, MCSD12, RMM10, RMM14a], code changes [KNGW13, CCLVAP14, LVCCAP15, JM17, MBDP+14, MBP+17], database usages [LVLVP16], release notes [MBDP+14, MBP+17], user reviews [DSPA+16], user stories [KJM17], activity diagrams [BH11] or source code exceptions [BW08].

It is worth mentioning that discussed approaches rely on different techniques. For instance, some of these approaches rely on information retrieval (IR) techniques and algorithms such as VSM (*e.g.,* [HAMM10]), PageRank [LM11] (*e.g.,* [MM14, MM16]), LexRank [ER04] (*e.g.,* [TR16]), or Maximal Marginal Relevance (MMR) [BEBTM08] (*e.g.,* [TR16]). However, some summarization approaches leverage new techniques, for instance, HoliRank [PSB+17, PML15], an extension of LexRank devised to analyze data holistically by considering the heterogeneous nature of information in software artifacts. Some other approaches rely on techniques such as neural networks (*e.g.,* [IKCZ16]), topic modeling (*e.g.,* [FCR+17, MLMW14]) and method/class stereotypes (*e.g.,* [MAS+13, MMPVS13, DCM06, DCM10, LVLVP18]).

At the end, although summarization is one of the most common techniques that can be adapted to document different types of software artifacts, they fall short if the information to comprehend the code is not inside the original document. In these situations, one needs to seek information using external sources. This leads us to the mining of crowd knowledge.

2.2.2 Mining Crowd Knowledge

The term “crowd knowledge” refers to any type of information produced by the crowd, in our scope, developers. Prominent examples of such type of knowledge are Stack Overflow discussions [Sta17], where a developer can find numerous source code associated with natural language descriptions. Relying on this fact, researchers have proposed a variety of approaches to mine and process crowd knowledge automatically, and distill the most relevant parts. Such approaches are mostly implemented in form of code search engines and recommendation systems, able to suggest different types of information. In the following, we elaborate on these studies.

API and code usage examples Automated extraction/recommendation of API usage examples is vastly studied by researchers. Such approaches [LSX18, HM05, SM06, MWH06, MBDP+15, SIH14, JAM+17, TR16, RC15, BW12, BOL10, MPG+13, PBDP+14, PSB+17] mine software repositories to find representative API usage examples for assisting developers when trying to use a class or method, or when finding code examples showing how to implement a given task/feature.

For instance, Moreno *et al.* [MBDP+15] present *MUSE*, an approach to automatically generate concrete usage examples of a given API mined from client projects using such an API. Buse and Weimer [BW12] proposed to generate documented abstract API usages by extracting and synthesizing code examples of a particular API data type. Glassman *et al.* [GZHK18] developed a visualization tool that mines API usage code examples regarding a given API and summarizes them with the goal of assisting developers in learning API usage.

Moreover, several techniques and tools have been proposed in the literature to support developers in using APIs. Many of them aim at creating code examples for an API of interest. In this line of research falls *MAPO*, the tool proposed by Xie and Pei [XP06] and extended by Zhong *et al.* [ZXZ+09]. *MAPO* can mine abstract usage examples of a given API method. *UP-Miner* [WDZ+13] is a variation of *MAPO* that removes the redundancy in the resulting example list.

Petrosyan *et al.* [PRDM15] proposed a text classifier-based approach to automatically retrieve tutorial sections explaining how to use a given API type, while Treude and Robillard [TR16] present an automated approach to augment API documentation with a complementary relevant piece of information discussed on *Stack Overflow* posts. They show their machine learning based approach surpasses existing techniques, *e.g.,* text summarization. Furthermore, Azad *et al.* [ARG17] present a technique to predict how API methods should be used by identifying co-changing API elements from the change history of open-source projects and *Stack Overflow* posts. Earlier, Dekel and Herbsleb [DH09] developed an IDE plugin, called eMoose, which augments API method invocation with usage directives extracted from Javadoc.

A different type of work is the approach by Robillard and Chhetri [RC15]. They present an automated approach developed as an IDE plugin, named *Krec*, to identify and retrieve the relevant piece of information in API reference documentation. *Krec* is able to categorize the text fragments in API documentation as indispensable, valuable, or neither, based on their semantic content.

Extracting specific fragments of video tutorials In a distinct work, Ponzanelli *et al.* [PBM+16a, PBM+16b] implement *CodeTube*[1], an approach to automatically extract software development video tutorials available on the web, and to return fragments related to a developer's query. For that, this approach analyzes video contents to extract information such as displayed code fragments, together with the speech information provided by audio transcripts.

Other useful information relevant to the source code at hand Needless to mention, mining-based approaches don't limit to above classes of information. For example, Ponzanelli *et al.* [PBL13, PBDP+14] present a novel approach to automatically retrieve a ranked list of *Stack Overflow* discussion relevant to the developer's query. They implemented this approach in form of an Eclipse plugin called *Seahawk*. *Seahawk* provides developers with both manual and automatic interactions. In the automatic form, it generates an automatic query based on the keywords found in the Java code entities shown in the current IDE window.

Automated documentation of a code snippet Closer to our line of research are techniques leveraging crowd source information to document a code snippet of interest. The automatic documentation of code based on examples from the crowd has been explored by Vassallo *et al.* [VPDPC14] and Rahman *et al.* [RRK15]. Both these works mine SO discussions to re-document a Java software system but at different granularities: method and code fragment, respectively. The approach by Vassallo *et al.* searches on Stack Overflow for sentences related to the method the developer is interested in documenting. For example, if the developer is working on the Apache Lucene project (one of the two systems used in the evaluation), and she wants to document a method implemented in a class, the approach searches in SO using as search keys project name + class name + method name. This means that this approach can only support the documentation of very well known systems widely discussed on SO, while it cannot support the documentation of unknown software projects, like mobile apps still to be published on the Google Play store. Concerning the work by Rahman *et al.* [RRK15], the authors present *CodeInsight*, a tool pioneering the mining of insightful comments about a snippet of code from SO. *CodeInsight* looks for comments discussing bugs or improvement tips for code.

Wong *et al.* [WLT15] exploit the use of existing code comments for automated code comments generation with an approach called *ColCom*. This approach mines a set of given projects (inputs) to generate code comments for a target project. The idea is to reuse existing code com-

[1] See `http://codetube.inf.usi.ch`

ments from input projects in order to generate comments for a target project. The results show that 23.7% of the automatically generated code comments are useful at describing the source code, which suggests an improvement on the earlier approach by the same authors, *AutoComment* [WYT13], in terms of number of useful comments generated for the same set of target projects in both research.

2.2.3 Summing Up

The mentioned state-of-the-art approaches vary mostly on the way information is retrieved, processed and finally presented to developers. The source code on which a developer is working is the primary input to most of them, regardless of other parameters that vary from person to person or task to task, and can be taken into account for more relevant and useful results. Binkley *et al.* [BLH+13] stress the importance of "useful" documentation, besides mere "good" documentation, and emphasize the necessity of exploiting non-code factors, such as the expertise level of the developer, when generating documentation. Moreover, Happel *et al.* [HM08] conducted a survey on some well-known recommender systems and discussed their limitations and extant challenges. Their findings indicate that obtaining useful documentation is not a straightforward task and there are several challenges to be addressed.

Despite these efforts, automated documentation is still an emerging research area, far from being mature. In a recent proposal by Robillard *et al.* [RMT+17], a review of state-of-the-art approaches to enable automated on-demand documentation has been conducted and the key challenges are outlined in three categories, demonstrating potential research avenues in software documentation: information inference (*i.e.,* mechanisms to model and infer information), document request (*i.e.,* mechanisms to enable developers to express their information needs in a better way) and document generation (*i.e.,* approaches to generate appropriate output).

In the context of automating developer documentation, we developed ADANA [ABLVL19], a novel approach to automatically generate and inject comments that describe a given piece of Android related code. We elaborate on this work in Chapter 4. The closest work to ADANA, in terms of granularity, is the one by Ying *et al.* [YR13, YR14], which also focuses on automatic documentation of source code snippets, however, limited only to extractive summarization. As opposed to that, ADANA "freely" documents a snippet with descriptions mined from the Web.

As discussed earlier, Wong *et al.* [WLT15] also exploits the use of existing code comments for automated code comment generation with an approach called *ColCom*. Their approach does only work with type-1 and 2 clones found in a limited set of projects given as an input to the approach. Unlike ADANA, *ColCom* does not rely on a central persistent database which is continuously updated over time, and its performance heavily depends on the input projects. Moreover, since the code comments are extracted from source code, *ColCom* relies on a set of heuristics to identify the code comments associated to a code snippet.

2.3 Summary and Conclusion

In this chapter we reviewed the state of the art with respect to software documentation, focusing on two major lines of research related to our work, namely (i) developing tools and approaches to automatically generate or recommend documentation, and (ii) empirically investigating different documentation aspects and their impact.

As seen, many of previous empirical studies on documentation issues are conducted through interviews and survey which means they are limited in terms of population and diversity. We address this limitation in Chapter 5 presenting our large-scale qualitative study on software documentation

issues [ANVM+19]. Moreover, we also learnt that although previous studies have studied documentation issues and information types, there is still a gap when determining which are the more relevant ones to practitioners. We fill this gap with our other study in Chapter 6 where we conduct two surveys with software developers and professionals.

Moreover, we found out that automated documentation approaches are still encountering many roadblocks [RMT+17]. In this regard, we made our first attempt and present ADANA, a novel approach to automatically generate and inject comments that describe a given piece of Android related code [ABLVL19]. We elaborate on this work in Chapter 4.

In the following chapters in Part II, we examine our contribution and discuss our findings.

Part II

Software Documentation: Automation and Challenges

The usage of Application Programming Interfaces (APIs) is an integral part of software development, and it strongly influences how developers build their applications. The design of an API is particularly important, and previous studies have shown the importance of APIs, and at the same time, their issues. In this context, we investigated the impact of poorly documented APIs on their client projects (i.e., the projects using such APIs). In ***Chapter 3*** *we present our large-scale study on 75 popular Java libraries and their 14 thousand client projects Our statistical analysis indicated the negative impact of poor documentation, and underlined the need for good documentation in software systems.*

To partially address the lack of good documentation, we decided to move towards automated generation of documentation approaches. In ***Chapter 4****, we present ADANA, a novel approach to automatically generate and inject comments that describe a given piece of Android-related code. Our evaluation studies showed the potential benefits of our tool in code comprehension activities, both in terms of time needed to comprehend a given piece of code and of the reached comprehension level. ADANA, nevertheless, comes with limitations. While studying the literature to figure out how to overcome these limitations, we concluded that there is a lack of empirical knowledge about documentation issues faced by software developers.*

This observation triggered our next two studies, devoted to understanding the nature of software documentation issues. ***Chapter 5*** *presents the first study which led to a detailed taxonomy of 162 types of issues faced by developers and users of software documentation. While our taxonomy was promising, it had not been validated by practitioners, making it mostly an academic construction without the much needed reality check. To address this shortcoming, in* ***Chapter 6*** *we performed a survey with practitioners to learn more about the significance of such issues. Additionally, we studied the types of information which are more important to developers in different contexts.*

3

A Large-scale Empirical Study on Linguistic Anti-patterns Affecting APIs

THE CONCEPT of monolithic stand-alone software systems developed completely from scratch has become obsolete, as modern systems nowadays leverage the abundant presence of Application Programming Interfaces (APIs) developed by third parties, which leads on the one hand to accelerated development, but on the other hand introduces potentially fragile dependencies on external resources.

In this context, the design of any API strongly influences how developers write code utilizing it. A wrong design decision like a poorly chosen method name can lead to a steeper learning curve, due to misunderstandings, misuse and eventually bug-prone code in the client projects using the API. It is not unfrequent to find APIs with poorly expressive or misleading names, possibly lacking appropriate documentation. Such issues can manifest in what have been defined in the literature as Linguistic Antipatterns (LAs), *i.e.,* inconsistencies among the naming, documentation, and implementation of a code entity. While previous studies showed the relevance of LAs for software developers, their impact on (developers of) client projects using APIs affected by LAs has not been investigated.

In this chapter we fill this gap by presenting a large-scale study conducted on 1.6k releases of popular Maven libraries, 14k open-source Java projects using these libraries, and 4.4k questions related to the investigated APIs asked on Stack Overflow. In particular, we investigate whether developers of client projects have higher chances of introducing bugs when using APIs affected by LAs and if these trigger more questions on Stack Overflow as compared to non-affected APIs.

Structure of the Chapter

- **Section 3.1** provides the motivation for this chapter.
- **Section 3.2** introduces the concept of Linguistic Antipatterns on which this chapter is based.
- **Section 3.3** presents the study design, while our findings are discussed in **Section 3.4**.
- **Section 3.5** discusses the threats that could affect the validity of our results.
- **Section 3.6** concludes this chapter.

Accomplishments in a Nutshell

A Large-scale Empirical Study on Linguistic Antipatterns Affecting APIs [ANBL18]
Emad Aghajani, Csaba Nagy, Gabriele Bavota, Michele Lanza
In *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*, pp. 25–35. IEEE, 2018.

3.1 Motivation

The usage of Application Programming Interfaces (APIs) is an integral part of software development, and it strongly influences how developers build their applications. For instance, it has been shown that the stability of a software system highly depends on the libraries it uses [KMS15], or that placing a method in the right API class can significantly speed up development, even up to an order of magnitude [SM08].

The design of an API is particularly important, and previous studies investigated what makes an API *usable* or *maintainable* [MS16, MRTS98, Var16]. Books have been written about this topic [Tul08, Blo18], and developers can refer to guidelines or "best practices" prepared for these purposes [Red11, swi18]. The design of an API directly affects its usage [ANN+17] and learning curve [Rob09]. In such a context, factors playing a role include, but are not limited to, naming, encapsulation, object-oriented design, explicitness of pre/post-conditions, and updated documentation. On this last point, Robillard and Deline have identified API documentation as the main source of learning obstacles for developers [RD11]. It has also been shown that – despite many APIs being actively maintained and updated –, these documents are also prone to mistakes and inconsistencies, due to the high cost of keeping them updated and in sync with changes [ZS13, ZGC+17].

Duala-Ekoko and Robillard [DER12] have shown that developers rely on the API names when the documentation is missing or is incomplete. However, assigning good names to API methods is not an easy task [PFM13] and poorly chosen names can lead to problems later. In this context Arnaoudova *et al.* [ADPA16] formalized issues affecting the design and documentation of code components, presenting a catalog of 17 *Linguistic Antipatterns* (LAs), representing inconsistencies among the naming, documentation, and implementation of an entity. The authors showed that LAs are perceived negatively by developers since they hinder program comprehension. A recent study [FMAA18] using Near Infrared Spectroscopy to observe the cognitive load of 70 undergraduate and graduate students working with code snippets found that the presence of LAs significantly increases the cognitive load of developers.

Given the importance of *comprehensibility* and *usability* in the context of APIs, we conjecture that APIs affected by LAs can be problematic for client projects using them. To validate our conjecture we performed a large-scale study to investigate:

- *The impact of LAs affecting APIs on the likelihood of introducing bugs in the client projects using the API.* We analyze 1.6k releases of 75 popular Maven libraries exposing a total of 1.6M unique API methods and 14k client Java projects using them. We use the LA detection tool by Arnaoudova *et al.* [ADPA16] to identify LAs affecting the 1.6M APIs. For each client project C_i using a set of APIs A_{C_i} provided by the considered libraries, we mine the commits in C_i introducing the first usage of each API in A_{C_i}. Finally, using the SZZ algorithm [SZZ05], we identify bug-inducing commits and compare the likelihood of introducing a bug in the client project when using for the first time an API affected/not-affected by LAs.

- *Whether developers tend to ask more questions on Stack Overflow about APIs affected by LAs.* This would be an indication of higher difficulties experienced by developers in comprehending and adequately using APIs affected by LAs. We analyzed 4.4k Stack Overflow questions in which one of the API methods provided by the 75 Maven libraries is explicitly mentioned. We compare the proportion of questions asked for APIs affected/not-affected by LAs.

We quantitatively and qualitatively analyze our data. While our statistical analysis suggests that when using an API for the first time, developers of the client projects have a 29% higher chance of introducing a bug if such an API is affected by a LA, our qualitative investigation highlighted no

influence of LAs on the likelihood of introducing bugs in the client project. Similarly, we found no evidence that LAs affecting APIs trigger *Stack Overflow* questions. Our findings, besides providing a different perspective on the impact of LAs on code-related activities, also emphasize the need for combining both quantitative and qualitative findings in this type of observational studies.

3.2 Source Code Linguistic Antipatterns

In this section we introduce LAs following the seminal work of Arnaoudova *et al.* [ADPA16] who first studied linguistic antipatterns in source code. Arnaoudova *et al.* present a catalog of 17 LAs capturing inconsistencies among the naming, documentation, and implementation of attributes and methods. They showed that LAs are negatively perceived by developers who highlighted their negative impact on code comprehension. They also released a tool for detecting LAs in Java code[1]. In this chapter, we focus on the 12 LAs related to methods, presented in Table 3.1, since we aim at investigating their impact on (developers of) client projects using APIs affected by such LAs. These 12 antipatterns are classified into three categories (A, B, and C) that we present here through demonstrative examples. For further details, we refer the interested reader to Table 1 in [ADPA16] for a complete description of the LAs accompanied by real examples found in open-source projects. In the catalog, each type of LA is identified with an ID (*e.g.,* A.1 is the first LA belonging to the A category). We use the same IDs to ease the mapping between Table 1 in [ADPA16] and this chapter.

Table 3.1. LAs related to methods introduced by Arnaoudova *et al.* [ADPA16]

Category	ID	Description
(A) do more than they say	`A.1`	"Get" - more than accessor
	`A.2`	"Is" returns more than a boolean
	`A.3`	"Set" method returns
	`A.4`	Expecting but not getting single instance
(B) say more than they do	`B.1`	Not implemented condition
	`B.2`	Validation method does not confirm
	`B.3`	"Get" method does not return
	`B.4`	Not answered question
	`B.5`	Transform method does not return
	`B.6`	Expecting but not getting a collection
(C) do the opposite than they say	`C.1`	Method name and return type are opposite
	`C.2`	Method signature and comment are opposite

Category A: do more than they say. This category includes four LAs (A.1 - A.4) related to methods that do more than what their signature and documentation indicate. For example, *A.1 "Get" - more than accessor* identifies getter methods which do actions other than returning the corresponding attribute without documenting it [ADPA16].

Category B: say more than they do. Includes five LAs (B.1 - B.6) related to methods doing less than what their signature/documentation says. For instance, *B.1 Not implemented condition* affects methods in which the comment suggests a conditional behavior not implemented in the body [ADPA16].

[1] See `http://www.veneraarnaoudova.com/linguistic-anti-pattern-detector-lapd/`

Category C: do the opposite than they say. This category includes two LAs (C.1 and C.2) affecting methods implementing behavior that is the opposite as compared to the one suggested by their signature and comments. For example, *C.1 Method name and return type are opposite* identifies methods having a name that is in contradiction with their return type (*e.g.,* a method named `disable` having `ControlEnableState` as return type) [ADPA16].

3.3 Study Design

The *goal* of the study is to investigate (i) the impact of LAs affecting APIs on the likelihood of introducing bugs in the client projects using the API, and (ii) whether developers are more prone to ask questions on Stack Overflow when the APIs are affected by LAs.

The *context* is represented by 1.6k releases of 75 Maven libraries, 14k client projects using those libraries, and 4.4k Stack Overflow questions. The *quality focus* is on APIs source code quality and comprehensibility that might be negatively affected by the presence of LAs.

3.3.1 Research Questions

Our study addresses the following two research questions:

RQ_1. *What is the impact of the LAs affecting APIs on the likelihood of introducing bugs in the client project?* Here with "client project" we refer to the project using the API. We conjecture that the presence of LAs in the APIs can create issues to the developers of the client projects that might misinterpret the API and introduce bugs when using it. Indeed, previous studies showed the negative impact of LAs on the comprehensibility of the affected code components [ADPA16]. Note that we do not limit our analysis to the comparison of APIs affected and not-affected by LAs, but we also investigate how each of the 12 different method-related LAs defined by Arnaoudova *et al.* [ADPA16] increases the likelihood of introducing bugs when using APIs affected by it.

RQ_2. *Are APIs affected by LAs more likely to trigger discussion on Stack Overflow?* This research question aims at verifying whether LAs trigger more questions from developers using the affected API methods. As for RQ_1, we also report the types of LAs triggering more questions.

3.3.2 Context Selection

To answer our research questions the first step is the selection of the Java libraries to analyze, and their client projects. We limit our study to Java since, as we stated in Section 3.2, the tool we use to detect LAs only supports Java code.

Due to the need of automatically identifying the client projects of a given library, we decided to focus our study on Maven libraries. Indeed, client projects interested in using a Maven library simply define a `pom.xml` file to specify the libraries they want to use. We selected all libraries belonging to the four most popular Maven categories[2]: *Testing Frameworks* (45 libraries and 1,103 releases), *Logging Frameworks* (38 and 997), *Core Utilities* (5 and 248), and *JSON Libraries* (67 and 1,369). The number of mined releases excludes non-final-release versions such as *beta, release candidate, etc.* since APIs might be not yet finalized in those versions.

[2] See `https://mvnrepository.com/open-source`

Overall, we collected 3,708 release versions of selected libraries and we mined GitHub to identify their client projects. Using the GitHub search API, we first identified in GitHub all Java projects having at least one `pom.xml` file, needed to declare dependencies toward Maven libraries. This resulted in the identification of 17,659 client projects, using 118,626 `pom.xml` files and declaring ∼1.1M dependencies in total. We downloaded all the identified pom files and converted them into a standard format. This is needed since it is possible to use variables in pom files, or to declare dependencies using version intervals or relative version schemas (*e.g.,* declaring a dependency towards the `latest` version of a library). Since we need to know the exact version from which the client project depends on, we used the `mvn help:effective-pom` command to preprocess the pom files and obtain dependencies with their absolute version numbers.

Finally, we excluded all `pom.xml` files not reporting any dependency towards one of the 3,708 library releases subject of our study. This left us with 14,743 client projects.

Table 3.2. Maven libraries and client projects considered

Category	#Libraries	#Releases	#Client Projects
Testing Frameworks	25	268	13,169
Logging Frameworks	19	304	8,732
Core Utilities	5	175	7,343
JSON Libraries	26	545	6,703
Total	**75**	**1,642**	**14,743**

Once collected the ∼14k client projects, we excluded from our study all library releases for which we did not identify any client project.

Indeed, client projects are needed to answer RQ_1, and we preferred to have a consistent dataset for both research questions. This decreased the number of libraries considered in our study to 75 for a total of 1,642 releases. Table 3.2 shows the number of libraries and releases we consider for each of the four popular Maven categories as well as the number of client projects identified for them.

3.3.3 Data Extraction

This section describes the data extraction process we followed to answer our research questions.

Parsing the libraries and the client projects, and identifying LAs

We downloaded the source code of the 1,642 library releases by using the `mvn dependency:sources` command. Then, we used the *Eclipse JDT Parser* to parse the code of each library to extract all the method declarations creating a database of 1.6M public (*i.e.,* API) and 800k private methods. When only considering the latest release of each of the considered libraries (used for RQ_2 as well), these numbers drop to 57k and 29k for public and private methods, respectively. Besides that, during the parsing process we also extracted precise type information related to the fully qualified class name of the method parameters' type, the return type, and the class defining each method. This information is needed to accurately (i) identify API invocations in the client projects (needed for RQ_1) and (ii) link Stack Overflow questions to library APIs (needed for RQ_2).

To identify the LAs affecting the APIs, we exploited the tool by Arnaoudova *et al.* [ADPA16] and able to detect the linguistic antipatterns described in Section 3.2.

RQ_1-specific data extraction

Similarly to what was done for libraries, we used the *Eclipse JDT Parser* to parse the 14k client projects and extract from them a total number of 96M invocations together with their precise type information. In particular, given the `jar` files of the libraries the client project depends on, the parser is able to bind the invoked methods to their original declaration and extract type information regarding the parameters, return type and the class whose the method belongs to. Having available the fully qualified class name of the class defining the invoked method, it is possible to differentiate between local and non-local invocations. We mark as local invocations (and exclude them since irrelevant for our study) all those related to methods declared in classes having one of the client project's packages in their fully qualified name. From the remaining non-local invocations, we exclude the ones related to methods belonging to classes from the `java.*` packages.

Finally, we compare all the remaining non-local invocations with the APIs declared in the library versions the client project depends on (excluding libraries not considered in our study). Such a matching is precise thanks to the fact that we consider the complete method signature, the fully qualified names of the types of its parameters, its return type, and of the class declaring it.

The collected method calls from the client projects to the libraries are necessary but not sufficient for answering RQ_1. Indeed, our goal is to compare the likelihood of introducing a bug in the client project when using for the first time an API affected/not-affected by LAs. To this aim, we also need to identify (i) the exact commit in which each API used by each client project has been introduced for the first time in its code, and (ii) the bug-introducing commits, meaning commits that likely induced a bug-fixing activity. This way we can count when the use of an API for the first time (affected/not-affected by LAs) resulted in the introduction of bugs.

We used the `git log -L` l_n,l_n:F_{path} command to identify for each API invocation in the client projects the commit in their change history in which they have been introduced for the first time. In the command, l_n indicates the line number in which the method invocation is present in the client's code, and F_{path} is the path of the client's file containing the invocation. The command traces back the commit history of the source code at the given line, and we took the commit where the API invocation was first added to the codebase.

To identify bug-fixing activities performed during the change history of the client projects, we used an approach proposed by Fischer *et al.* [FPG03], *i.e.,* by mining regular expressions containing issue IDs and the keyword "fix" in the commit notes, *e.g.,"fixed issue #ID"* or *"issue ID"*. Then, we identify commits that introduced bugs[3] by using the SZZ algorithm [SZZ05], which is based on the annotation/blame feature of versioning systems. In summary, given a bug-fix commit, k, the approach works as follows:

1. For each file f_i, $i = 1 \ldots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).

2. Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of git is used to identify the file revision where the last change to that line occurred. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \ldots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

Matching the commits in which APIs have been introduced for the first time and those that introduced bugs will allow us to answer RQ_1 through the data analysis described later.

[3]The right terminology is "when the bug induced the fix" because of the intrinsic limitations of the SZZ algorithm, which cannot precisely identify whether a change actually introduced the bug.

RQ_2-specific data extraction

We mine the official Stack Overflow dump released in June 2017 to identify all questions explicitly mentioning an API method from the latest release of one of the 75 considered libraries. Such an analysis is limited to the latest library releases since API versions are rarely explicitly mentioned in the posts.

For this analysis, we implemented an approach to extract the qualified names of methods referenced in the code blocks of *Stack Overflow* questions. Existing approaches typically look for class names of APIs mentioned in the text, code block or `href` markup links of Stack Overflow [KPG+13, PTG12, TR16]. However, we need an approach which can link *Stack Overflow* questions to exact API methods, also considering their parameters.

First, we extract code blocks from Stack Overflow posts tagged with the Java tag, we parse these code blocks with the srcML infrastructure [MC15], and then we collect the method signatures for method invocations with an algorithm which runs on the AST of a code block provided by srcML. For parsing, we have chosen srcML as a lightweight and robust parser, which can tolerate the usually incomplete source fragments on *Stack Overflow* but provides the necessary information for our analysis. Here, we have to be prepared for sample code snippets with often missing `import` statements or even class or method declarations. In addition, developers tend to use code blocks sometimes only for formatting purposes, *e.g.,* to emphasize numbers or sample commands sometimes written in other languages. To avoid these, we filtered code blocks shorter than 20 characters. After the extraction of code blocks, our algorithm collects the type information (*i.e.,* class name) of declaration nodes in the AST and for method invocations on local/instance variables, it pairs the method name and number of arguments with the type information of the related variable. If it cannot find the referenced variable, it handles the reference as a static reference. As a result, for each code block we have a set of `className,methodName,numberOfArguments` tuples describing all method invocations in the code block. The extracted method references are stored in a database along with the API method declarations and they are linked to each other.

We analyzed 1,269,994 questions in Stack Overflow having a total number of 2,071,992 code blocks. After the parsing step, the collection of class and method name pairs provided us 804,104 unique tuples and a total number of 3,308,072 method references.

Knowing the Stack Overflow questions referencing each specific API will allow us to answer RQ_2 by verifying whether APIs affected by LAs trigger more questions from developers.

3.3.4 Data Analysis

To answer RQ_1, we compare the likelihood of introducing a bug in the first commit introducing in the client projects APIs affected and not-affected by LAs. In particular, we compute the following four groups:

- $\textbf{ANB}_{Clean}$ indicating the number of commits introducing for the first time an API not affected by any LA that do not induce a bug;
- $\textbf{AB}_{Clean}$ indicating the number of commits introducing for the first time an API not affected by any LA that induce a bug;
- $\textbf{ANB}_{LA}$ indicating the number of commits introducing for the first time an API affected by a LA that do not induce a bug;
- $\textbf{AB}_{LA}$ indicating the number of commits introducing for the first time an API affected by a LA that induce a bug;

Then, we use Fisher's exact test [She03] to test whether the proportions of $\text{AB}_{Clean}/\text{ANB}_{Clean}$ and $\text{AB}_{LA}/\text{ANB}_{LA}$ significantly differ. In addition, we use the Odds Ratio (OR) [She03] of the two proportions as effect size measure. An OR of 1 indicates that the condition or event under study (*i.e.,* the chances of inducing a bug) is equally likely in two compared groups (*e.g.,* clean *vs* LA). An OR greater than 1 indicates that the condition or event is more likely in the first group (that, in our analysis, will be LA). On the other hand, an OR lower than 1 indicates that the condition or event is more likely in the second group (Clean).

We also perform the same analysis when considering specific types of LAs. Meaning that, for each of the LA_i types we detected in our dataset, we compute the groups $\textbf{ANB}_{LA_i}$ and $\textbf{AB}_{LA_i}$ and again compare the proportion $\text{AB}_{LA_i}/\text{ANB}_{LA_i}$ with that of the $Clean$ group, with the goal of identifying what the most "dangerous" LAs are (if any).

To answer RQ_2, again we rely on the Fisher's exact test and on the odds ratio to verify whether developers tend to ask more questions about APIs affected by LAs as compared to clean APIs. We also compare the distributions representing the number of Stack Overflow questions triggered by APIs affected and not-affected by LAs. We use the Mann-Whitney test to compare the two distributions [Con98] with results intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [GK05]. We follow well-established guidelines to interpret it: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [GK05].

Finally, we qualitatively analyze our findings in both research questions.

3.4 Results & Discussion

Before answering our research questions, we start by describing our dataset from different perspectives in order to give the reader a complete view of the subject APIs, client projects, and LAs.

Out of 2.4M methods defined in the 1,642 Maven releases we analyzed, 1.6M methods (66.0%) are public (*i.e.,* API methods). When only considering the latest release of the 75 subject libraries, the percentage of public methods is stable at 66.3%. Thus, although an API should be as minimal as possible to avoid revealing unnecessary details [Hen07], the studied Maven libraries expose a high number of public methods (*i.e.,* API methods).

We also inspected the presence of Javadoc documentation in the 2.4M methods. We used the *Eclipse JDT Parser* to detect comments using the Javadoc syntax (*i.e.,* `/** ... */`) right before a method declaration. We found that 22.5% of methods have a Javadoc comment, with such a percentage increasing to 46.2% when only focusing on public methods. These percentages are quite stable when only considering the latest release of each library (48.6% for public methods and 23.3% for all methods). While the higher Javadoc coverage for public methods as compared to private methods is a quite expected results (since these are the methods client projects are supposed to use), we still found that more than half of public methods are not documented through Javadoc. A possible explanation for this finding could be that many public getter and setter methods present in the studied libraries are not documented since, in many cases, their code is self-explanatory. We verified such an explanation by computing the number of public getter and setter methods in the set of 2.4M methods and by verifying how many of them are not documented. Overall, we found 518k getters and setters (406k getters and 111k setters), 189k of which (148k getters and 41k setters) documented (36%). Thus, excluding getters and setters from the counting, we still have 49% of uncommented public methods, do not substantially changing our finding.

Table 3.3 reports the LAs we found in our dataset. For each of the twelve LAs we considered, we report: (i) its ID (column "LA ID') allowing its mapping to Table 1 in the work by Arnaoudova

et al. [ADPA16]; (ii) its name, providing a short description of the type of issue it captures; (iii) the number of libraries and releases, among the ones we studied (*i.e.,* 75 libraries and 1,642 releases), in which we found at least one method affected by it; and (iv) the total number of methods affected by it. We report this information both when considering all methods ("Overall" in Table 3.3) as well as when only focusing on public methods.

Table 3.3. Number of libraries/releases/methods affected by LAs

LA ID	Overall			public (APIs)		
	#Libraries	#Releases	#Methods	#Libraries	#Releases	#Methods
A.1	34	669	4,160	29	564	2,680
A.2	16	348	6,337	15	330	5,977
A.3	32	697	5,680	23	593	4,029
A.4	37	853	5,846	33	822	4,459
B.1	41	831	10,418	38	803	9,539
B.2	30	621	2,170	20	371	561
B.3	17	424	1,147	13	293	781
B.4	10	282	1,463	6	118	1,148
B.5	12	136	624	9	112	276
B.6	27	634	2,491	25	586	2,080
C.1	6	44	63	4	30	49
C.2	40	909	5,549	35	771	3,690
	59	**1,078**	**43,778**	**56**	**1,047**	**33,633**

We found 43,778 methods out of 2.4M (1.8%) affected by LAs, with such a percentage growing to 2.1% when only focusing on public methods (33,633 out of 1.6M). What is more interesting is that 64% of the studied releases have at least one public method affected by a LA. Thus, knowing whether the LAs increase the likelihood of introducing bugs in the client projects and of triggering questions on *Stack Overflow* is worth investigating.

Moving to the client projects and their relationship with the libraries we found 96.8M method invocations in the 14,635 client projects: 53.9M (55.7%) are local invocations[4], 28.1M (29.1%) are related to Java APIs, 2.2M (2.2%) concern invocations to APIs belonging to the studied libraries, and the remaining 12.6M (13.0%) target APIs from other libraries.

Interestingly, we found that client projects only use a very limited subset of the public methods exposed by libraries. Considering all releases, we found that only 2.4% of public methods (38,246 out of 1,613,176) are used by at least one client project. Such a percentage grows to 7.0% (3,986 out of 57,369) when only focusing on public methods belonging to the latest releases of the analyzed libraries. This finding confirms what has been observed by Sawant and Bacchelli [SB17], who reported that a considerably small portion of an API is actually used by developers. Table 3.4 reports the top 10 library releases in terms of the percentage of their APIs used by at least one client project.

Another interesting observation derived from our dataset is that 83.9% of API methods used in client projects (the same percentage holds when considering all releases as well as when only focusing on the latest release) is accompanied by a Javadoc documentation. Such a percentage is much higher as compared to the percentage of all public methods having a Javadoc comment (*i.e.,* 51.0% in the best case scenario, when not considering getters and setters).

Although we did not dig further into this finding via qualitative analysis, these numbers clearly show a correlation between the presence of Javadoc comment in public methods and their usage in

[4]We discriminate between local and non-local invocations as described in Section 3.3.3.

Table 3.4. Top-ten releases in terms of percentage of public API methods used by their clients

Library Release			#used	#API	Percentage
groupId	**artifactId**	**version**			
net.minidev	json-smart	2.3	184	453	41%
org.hamcrest	hamcrest-core	1.3	89	252	35%
org.hamcrest	hamcrest-all	1.3	185	655	28%
com.googlecode.json-simple	json-simple	1.1.1	26	105	25%
junit	junit	4.12	344	1,369	25%
commons-lang	commons-lang	2.6	476	2,317	21%
org.slf4j	slf4j-api	1.7.25	87	439	20%
com.google.code.gson	gson	2.8.2	146	763	19%
com.unboundid.components	json	1.0.0	34	194	18%
com.esotericsoftware.minlog	minlog	1.2	4	31	13%

client projects. The problem here is the impossibility to define the direction of the causation. Indeed, we do not know whether the client projects actually tend to use documented APIs or, instead, are the developers of the APIs that tend to only document APIs they expect to be used by client projects. Such an investigation is part of our future research agenda.

3.4.1 RQ_1: What is the impact of the LAs affecting APIs on the likelihood of introducing bugs in the client project?

As explained in Section 3.3.4, we extracted for each client project: (i) the commit in which each API it uses has been added for the first time in its code, and (ii) its bug-introducing commits, meaning the commits identified by the SZZ algorithm as likely to have triggered a bug-fixing activity in the future.

Having this data, we computed the cardinality of the four sets AB_{Clean}, ANB_{Clean}, AB_{LA}, and ANB_{LA} (see Section 3.3.4 for their definition). When considering all the twelve types of LAs, we obtained the following cardinalities: AB_{Clean}=1980, ANB_{Clean}=54918, AB_{LA}= 122, and ANB_{LA}=2612, leading to a statistically significant (p-value = 0.007) odds ratio of 1.29. This means that *when an API call is introduced in a client project for the first time, the likelihood of introducing a bug is 29% higher if the API is affected by a linguistic antipattern*.

Table 3.5. Odds ratio by type of LA (significant results only)

ID	LA Name	AB_{LA}	ANB_{LA}	Ratio	p-value
B.1	Not implemented condition	36	521	1.92	0.000
B.4	Not answered question	13	187	1.93	0.031
B.5	Transform method does not return	7	50	3.88	0.003

We also performed the same analysis for the 12 types of LAs we considered, and report the results in Table 3.5 for the LAs for which we obtained a statistically significant odds ratio. The first thing that leaps to the eyes is that all the LAs substantially increasing the chance of introducing bugs belong to the "B category" of LAs. These LAs are related to methods that do less than what their signature/documentation says.

The *"B.1 - Not implemented condition"* LA affects methods in which the comment suggests a conditional behavior that is not implemented in the method's body [ADPA16]. For example, if the comment states "*Returns true if the balance is higher than 0, false otherwise*" but that does not implement any

if statement to check the balance is affected by this LA. For B.1 the odds ratio is 1.92, indicating that developers have 92% higher chance of introducing a bug when committing for the first time usages of APIs affected by this LA as compared to clean APIs. While the statistical analysis provides a quite bold message, we manually analyzed all 36 commits in which, according to our data, the B.1 LA induced a bug-fixing activity, to verify what the role played by the LA actually was.

We found that in none of the 36 analyzed commits the usage of the API affected by the LA was actually the trigger for the future bug-fixing activity. This is due to the fact that the LAs of type B.1 involved in the 36 commits, while not false positives according to the B.1 definition, are not harmful. Let us explain why with one representative example, the case of the `concat` method implemented in *com.google.guava* library. In the Javadoc comment of the `concat` method it is documented a conditional behavior: "`@Throws NullPointerException` if any of the provided iterators is null", and such a behavior is not implemented in the method body through a conditional statement verifying whether the iterators provided as parameters are null. This makes `concat` affected by the *"B.1 - Not implemented condition"* LA. However, the `concat` method invokes the `checkNotNull` method by passing to it the iterators. The latter method is the one implementing the conditional statement throwing a `NullPointerException` when needed. Clearly, detecting these cases is far from trivial, since it requires interprocedural code analysis, currently not supported by the LA detection tool we used. In this specific case the bug was introduced in the same commit in which an invocation to this API was added in the client project, but the bug was not due to a misuse of such API. Similar observations hold for the other 35 commits.

The *"B.4 - Not answered question"* LA affects methods having their name in the form of predicate (*e.g.,* `isValidURL`) but not returning a `boolean` [ADPA16]. For B.4 the odds ratio is 1.93, indicating that developers have 93% higher chance of introducing a bug when committing for the first time usages of APIs affected by this LA as compared to clean APIs. Also in this case our manual analysis did not highlight a direct effect of the LA on the bug introduction. The detection tool perfectly worked, and did not detect any false positive. The problem was in the specific context in which the LAs were detected. Indeed, all the B.4 instances involved in the bug-inducing commits were detected in methods from classes assisting in the validation of arguments. For example, the `isTrue` method from the `Assert` class of the *org.springframework* library has its name in the form of predicate but returns `void`. The reason is that, as documented in the Javadoc, this method "asserts a `boolean` expression, throwing an `IllegalArgumentException` if the expression evaluates to `false`". In such a context, while the B.4 LA clearly affects the method, it is unlikely to be harmful. All the bug-inducing commits we analyzed follows such a pattern, and did not play a direct role in the introduction of the bugs we analyzed.

Finally, the *"B.5 - Transform method does not return"* LA is the one exhibiting the highest odds ratio (3.88), indicating that developers have ∼4 times the chance of introducing bugs when working with APIs affected by B.5 as compared to clean APIs. This LA affects methods having a name suggesting the transformation of an object but not returning anything (as opposed to the expected transformed object) [ADPA16]. In this case, our qualitative analysis showed that all the bug-introducing commits were related to the usage, from different client projects, of the `toJson` method from the *com.google.code.gson*. This method actually returns `void` in the library releases involved in the bug-inducing commits, thus being classified as affected by the B.5 LA. However, `toJson` takes as one of its parameters a `writer` that, as documented in the Javadoc, represents the "*Writer to which the Json representation needs to be written*". In other words, while the transformation of the object to JSON does not result in a new object to be returned, the output of this transformation is written somewhere and well documented in the method. Also in this case, the LA did not look responsible for the bug introduction in the analyzed cases.

Summary for RQ$_1$: Our statistical analysis indicated that when introducing for the first time

APIs affected by LAs in the code base, developers have 29% higher chance of introducing bugs as compared to when using clean APIs. Such an effect is mostly due to three types of LAs, namely *"B.1 - Not implemented condition"*, *"B.4 - Not answered question"*, and *"B.5 - Transform method does not return"*. However, in our qualitative analysis we did not find any strong evidence of their negative impact on the likelihood of introducing bugs.

3.4.2 RQ_2: Are APIs affected by LAs more likely to trigger discussion on Stack Overflow?

We had to face a number of challenges when linking APIs to *Stack Overflow* questions. We found classes with the same names in multiple libraries and/or in the Java API. When looking for *Stack Overflow* questions mentioning these classes but not reporting a package import in the code block (as it is very frequent in *Stack Overflow* posts), it is not possible to identify precisely which class of which library is referenced at that location. For example, the `dbunit` library has an `InputStream` class in the `org.dbunit.util.Base64` package. Moreover, this class has a `read` method without parameters just like the `read` method of `java.io.InputStream`. We found 616 questions with a code block using the `InputStream.read()` method but without a reference to the library or the package name the method belongs to. Thus, it is impossible to determine whether the `dbunit` library or the Java API was referenced in these questions. For this reason, we filter out from the set of APIs to link to the *Stack Overflow* questions (i) all classes appearing with the same name as another class in the Java API and/or in another library; and (ii) all methods which appear with the same name and arguments in another class of another library. We found that 200 classes of the libraries appear with the same name in the Java 9 API (Java Platform, SE, and JDK) and 136 classes in the Java EE 7 API. We also found 7,291 methods appearing with the same name, declaring class and number of parameters in multiple libraries. In the end, we have 34,260 public methods of 5,261 classes in our dataset to investigate how LAs trigger discussions on Stack Overflow. Remember that in this investigation we only focus on the APIs present in the last release of the 75 subject libraries. These API methods were referenced in 4,464 questions on *Stack Overflow* including 135 questions related to LAs.

Table 3.6. Odds ratio of methods discussed in *Stack Overflow* Questions with/without Linguistic Antipatterns

$SO_{LA}/NoSO_{LA}$	=	39/716	=	0.0544	(a)
$SO_{Clean}/NoSO_{Clean}$	=	891/33,406	=	0.0266	(b)
OddsRatio	=	**(a)/(b)**	=	**2.05**	

To address RQ_2, we calculate the *odds ratio* of methods (not)mentioned in Stack Overflow questions and methods (not)affected by LAs. Table 3.6 shows the different method sets needed to calculate the odds ratio. As done in RQ_1, LA is the set of methods affected by Linguistic Antipatterns, while $Clean$ are methods not affected. SO are methods mentioned at least in one *Stack Overflow* question, and $NoSO$ are methods not mentioned at all. As a result, the *odds ratio* is 2.05 indicating that methods affected by LAs are twice more likely to trigger questions on *Stack Overflow* than clean methods.

When comparing the distribution of the number of *Stack Overflow* questions related to methods affected and not affected by LAs with the Mann-Whitney, the p-value turns out to be 0.249 which, at a $\alpha = 0.05$, indicates no significant difference (and a negligible effect size of -0.06).

We also investigated which LAs affect the APIs discussed in *Stack Overflow* questions. Table 3.7 shows the number of different LAs found in methods which were also mentioned in *Stack Overflow* questions. The major part of the questions is related to the A1, B7, C2 and B1 categories. Note that in Table 3.7 we report the total number of questions asked for the methods affected by LAs (*i.e.,*

Table 3.7. Linguistic Antipatterns found in methods with related questions on SO

LA ID	LA name	#Question
A.1	"Get" - more than accessor	21
A.2	"Is" returns more than boolean	4
A.3	"Set" method returns	3
A.4	Expecting but not getting single instance	10
B.1	Not implemented condition	20
B.4	Not answered question	10
B.7	Method does not return the corresponding attribute	32
C.1	Method name and return type are opposite	1
C.2	Method signature and comment are opposite	34

135), while in the analysis with odds ratio we considered the number of methods affected by LAs and linked to at least one Stack Overflow question (*i.e.,* 39). Lastly, the detailed list of libraries having methods affected by LAs and discussed on SO can be seen in Table 3.8.

Table 3.8. Libraries having methods affected by LAs and discussed on SO

GroupId	ArtifactId	Methods	Questions
com.google.guava	guava	5	27
org.codehaus.plexus	plexus-utils	3	19
org.springframework	spring-core	6	19
log4j	log4j	4	17
xmlunit	xmlunit	1	15
junit	junit	3	13
commons-lang	commons-lang	1	5
org.apache.logging.log4j	log4-core	3	5
com.fasterxml.jackson.core	jackson-databind	2	3
org.apache.logging.log4j	log4j-api	2	3
com.pivotallabs	robolectric	2	2
org.springframework	spring-test	2	2
org.testng	testng	1	2
com.fasterxml.jackson.core	jackson-core	1	1
com.google.code.gson	gson	1	1
com.jayway.restassured	rest-assured	1	1
org.httpunit	httpunit	1	1

We manually investigated all these 135 questions. Our approach to identifying methods in code blocks of *Stack Overflow* questions spotted the library and the correct method of the API for 106 questions. For the rest, five questions are not available online anymore on *Stack Overflow* (we relied on the last release of the *Stack Overflow* database dump from June 2017), and in the remaining 24 cases it found a method with the same name, parameters and a declaring class of another library. This means that the approach was successful in 82% of the methods manually inspected.

We also checked whether the questions were indeed closely related to the API methods. We found 50 cases closely related to the usage of the method in the API, while in other cases the method was part of the sample code, but the questions were about some other code components or were not related to how the API should be used. For instance, the `Stopwatch.stop()` method of `com.google.guava` appeared in 14 questions discussing some performance issues. The code samples in these questions measured time with `Stopwatch`, but they were not related to the usage of

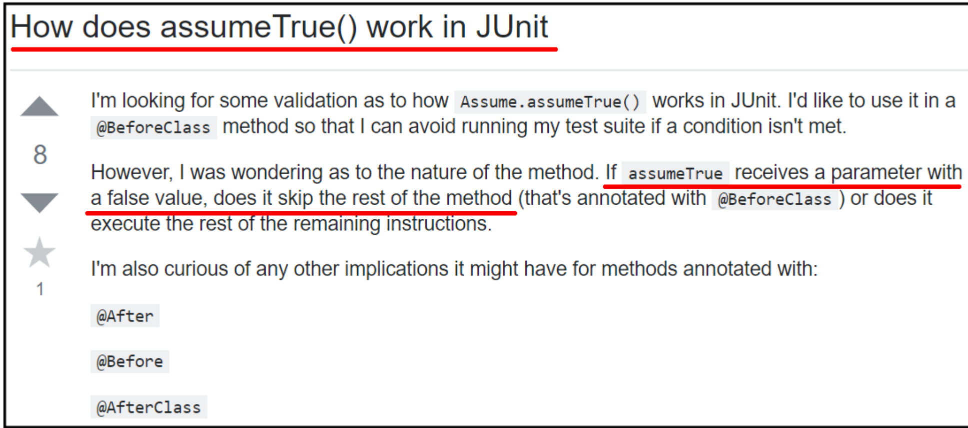


Figure 3.1. An example SO question related to the usage of `Assume.assumeTrue(boolean)`, a *junit* method with a LA because of an antonym in the documentation

`Stopwatch`.

Regarding the discussion of a problem related to the LAs, we found only three cases where the problem mentioned in the question could indeed originate from a problem related to the LA affecting the method. Even for these cases, this was not explicitly mentioned. An example can be seen in Figure 3.1. The `Assume.assumeTrue` method of *junit* has the following LA: "*C.2: Method comments and signature use antonyms: false versus true. Signature: Assume.assumeTrue(boolean b): void*". The reason for the LA is the documentation of the method, which says the following: "*If called with an expression evaluating to false, the test will halt and be ignored*". More interestingly, the `assumeFalse` has the following comment: *The inverse of assumeTrue(boolean).*" Although, the problem in the question is not explicitly related to the antonym in the documentation. Indeed, the documentation misses the information.

Summary for RQ$_2$: We did not find clear evidence that the existence of LAs admittedly triggers questions on *Stack Overflow*. We notice, however, that just like the example in Figure 3.1, some LAs are probably more prone to misunderstandings.

3.5 Threats to Validity

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. This is the most important kind of threat for our study, and is related to:

- *RQ$_1$: Approximations due to identifying bug-fixing commits using regular expressions [FPG03].* We used the approach proposed by Fischer *et al.* [FPG03] mining regular expressions in commit notes to identify bug-fixing commits, thus possibly identifying false positive and missing false negative commits.

- *RQ$_1$: Approximations due to identifying bug-inducing commits using the SZZ algorithm [SZZ05].* We used heuristics to limit the number of false positives, for example excluding blank lines from the set of bug-inducing changes. However, we are aware of possible imprecisions introduced by the SZZ algorithm especially due to tangled commits [HZ13] comprising a bug-fixing activity as well as other changes (*e.g.,* some refactoring operations).

- *RQ_1 and RQ_2: Accuracy of the LA detection tool*. To detect LAs we used the tools developed by Arnaoudova *et al.* [ADPA16]. Given the magnitude of our study, manually validating the output of the tool was clearly not an option. However, from the study reported in the original paper introducing the tool we used [APAG13], we know that the tool's precision for the twelve considered LAs is ∼77%. Moreover, our qualitative analysis helped in identifying some borderline LA instances impacting our findings.

- *RQ_2: Imprecisions in identifying Stack Overflow questions related to the investigated APIs*. Our approach to link *Stack Overflow* questions to methods of APIs relied on the extraction of method signatures from code blocks. This approach can miss cases when a method signature cannot be extracted from the code block or when it can be extracted, but the same signature appears in multiple APIs. This introduces imprecision in identifying questions. To estimate this imprecision, we manually investigated a sample set of 135 *Stack Overflow* questions which were related to methods with LAs. We observed a precision of 82%. However, due to a large number of questions tagged with Java, we could not estimate the recall of our approach, and we may miss questions related to APIs.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. This type of threats strongly affect the findings of both our research questions. For what concerns RQ_1, the bug introductions for commits related to APIs affected by LAs might be due to several factors totally unrelated to the presence of LAs, and similar observations hold for RQ_2. For this reason, we addressed internal validity by qualitatively analyzing our results.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures, and qualitative analysis.

Threats to *external validity* concern the generalization of results. In RQ_1 we studied a total of 1,642 releases from 75 popular libraries and their 14,743 client projects, thus ensuring a good generalizability of our results for what concerns Java libraries and client projects. In RQ_2 we limited our study to the latest release of each of the 75 considered libraries due to the need of linking *Stack Overflow* questions to API methods. For both research questions, larger replications of our study possibly performed by also including languages different than Java can help to confirm or contradict our findings.

3.6 Summary and Conclusion

In this chapter, we investigated the impact of Linguistic Antipatterns (LAs) affecting APIs on the developers of client projects using such APIs. We studied whether (i) developers are more likely to introduce bugs when using for the first time APIs affected by LAs as compared to clean APIs, and (ii) developers tend to ask more questions when working with APIs affected by LAs as compared to clean APIs.

While our statistical analysis indicated some effect of LAs on the likelihood of introducing bugs and of triggering Stack Overflow questions, our qualitative analysis did not allow us to explain such a phenomenon. Clearly, this does not contradict the strong empirical evidence showing the negative impact of LAs on code comprehensibility [ADPA16, FMAA18], nor the fact that LAs are considered as bad programming practices by software developers [ADPA16]. However, our findings call for additional investigation about the impact on LAs on code-related activities, maybe conducted through controlled experiments better allowing to isolate the effect of the studied variable.

Given the results of this study, we determined to focus on software documentation and, in particular, its automated generation. For that, in the next chapter, we present our first attempt for building an automated documentation tool, named ADANA.

4

Automated Documentation of Android Apps

DEVELOPERS do not always have the knowledge needed to understand source code and must refer to different resources (*e.g.,* teammates, documentation, the web). This non-trivial process, called program comprehension, is very time-consuming. While many approaches support the comprehension of a given code at hand, they are mostly focused on defining extractive summaries from the code (*i.e.,* on selecting from a given piece of code the most important statements/comments to comprehend it). However, if the information needed to comprehend the code is not there, their usefulness is limited.

In this chapter, we present ADANA, an approach to automatically inject code comments describing a given piece of Android code. Our approach reuses the descriptions of similar and well-documented code snippets retrieved and processed from *GitHub Gist* and *Stack Overflow*, two commonly used resources for developers. This enables our approach to improve its suggestions automatically over time. Moreover, our approach benefits from developers' feedback to improve itself.

We have evaluated ADANA extensively through three studies, each assessing its different aspects. Our evaluation has shown that our tool is able to aid the program comprehension process. In particular, the achieved results demonstrate that developers not only obtain a better understanding of the method under analysis when comments injected by our approach were present, but they also save time spent in the comprehension process.

Structure of the Chapter

- **Section 4.1** provides motivation for this chapter discussing state-of-the-art techniques and their limitations.
- **Section 4.2** details our approach and steps behind it, and **Section 4.3** explains the evaluation studies conducted.
- **Section 4.4** discusses the results we obtained, and address our most important research question whether ADANA helps developers during code comprehension activities performed on Android apps.
- **Section 4.5** presents the threats that affect the validity of our work, and finally **Section 4.6** draws our conclusions.

Supplementary Material

All the data used in this chapter as well as our tool ADANA are publicly available. More specifically, we provide the following items:

- **Replication Package.** [ada19c] The replication package includes the following material:
  - **Tool.** The ADANA Android Studio plugin
  - **Experimental Settings.** The scripts used in different steps of our framework, The configuration used for Simian clone detector, and two sets of stopwords used by the ASIA clone detector.
  - **Data.** The result of three studied conducted for evaluating ADANA and ASIA (see Section 4.4)
- ADANA **Website.** [ada19b] The website provides quick access to the latest release of our plugin, as well as links to the replication package and the paper itself. It also demonstrates how one can use ADANA to generate code comments for a sample code snippet.
- ADANA **Source Code.** [ada19a] The source code of the ADANA Android Studio plugin.

Accomplishments in a Nutshell

Automated Documentation of Android Apps [ABLVL19]
Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza
In *IEEE Transactions on Software Engineering*, To be published

ADANA **Android Studio plugin**
The implemented of our approach, ADANA, in form of an IDE plugin. Read more in Section 4.2.2. The plugin is available at `https://adana.si.usi.ch/`

ASIA **Clone Detector**
An approach built on top of standard IR clone detection and tailored for identifying clones in Android-related code. Read more in Section 4.2.2.

4.1 Introduction

Software developers do not always possess the knowledge needed to comprehend the source code they are handling. This is especially true when code lacks documentation and comments [Spi10], or when code and documentation do not co-evolve [FWG07, FWGG09, LVLVP15]. To make up for the lacking knowledge, developers often refer to teammates and other sources of information found on the Internet [LVD06], such as Q&A websites like *Stack Overflow*. However, what developers often obtain are higher-level pieces of information, which are certainly useful, but do not always help to answer the question of what a specific chunk of source code is doing. This question has been tackled by automated summarization approaches [MAS$^+$13, YR13, FCR$^+$17, CCLVAP14, YR14, MM16, LVLVP16, LVLV$^+$16], and by creating extractive or abstractive summaries [HAMM10]. While in the former a subset of code/comment elements is selected from the code chunk to describe it, the latter includes information which is not explicit in the original document [HAMM10]. However, if the information to comprehend the code is simply not there, these approaches fall short.

We present ADANA, an approach to automatically generate and inject comments that describe a given piece of Android-related code. Our approach reuses the descriptions of similar and well-documented code snippets and is powered by a knowledge base of 64k well-described code snippets automatically retrieved and processed from *GitHub Gist* and *Stack Overflow*. Our approach benefits from ASIA, a clone detector we tailored to identify Android-related code clones.

We evaluated ADANA in three studies. Results show that (i) ADANA can, on average, automatically document with code comments one third of the code composing a mobile app; (ii) the ASIA clone detector can find similar code snippets with good precision (∼70%); and (iii) the comments injected by ADANA help in code comprehension both in terms of time needed and comprehension level.

4.2 ADANA

ADANA is implemented as a framework that includes an Android studio plug-in, a set of backend services for analyzing and extracting data from online repositories, and a knowledge base for storing snippets and descriptions. ADANA works as depicted in Figure 4.1. Dashed arrows represent dependencies (*i.e.,* ① and ⑦), full arrows indicate flows of information between components. Black arrows (*i.e.,* ① to ④) indicate operations performed with the goal of building the ADANA knowledge base; red arrows represent actions triggered by a request to document a selected piece of code through the ADANA Android Studio plug-in.

ADANA mines from the Web pairs composed of code snippets related to Android development and their description, which illustrates the task/feature implemented through a snippet (①). Quality checks ② are performed on the mined data to remove noise, such as pairs including non-Java code or unlikely to contain a meaningful description (*e.g.,* a single word description). The selected pairs are provided to the *description standardizer* ③ to convert the mined descriptions into a format suitable to document the related snippet of code and to store the processed pairs in the ADANA database ④. This database serves as the knowledge base.

The developer using the ADANA Android Studio plug-in can select any snippet of code in the IDE and ask ADANA to describe it ⑤. The developer can also tune the "granularity level" of the description she desires (*e.g.,* describing every single block in the selected code, or getting an overall description of it). ADANA looks in the knowledge base for clones of the code snippet selected in the IDE by using ASIA, short for "Android SImilarity Assessment".

ASIA is a clone detection approach tailored for Android ⑥ - ⑧ that we have designed to support

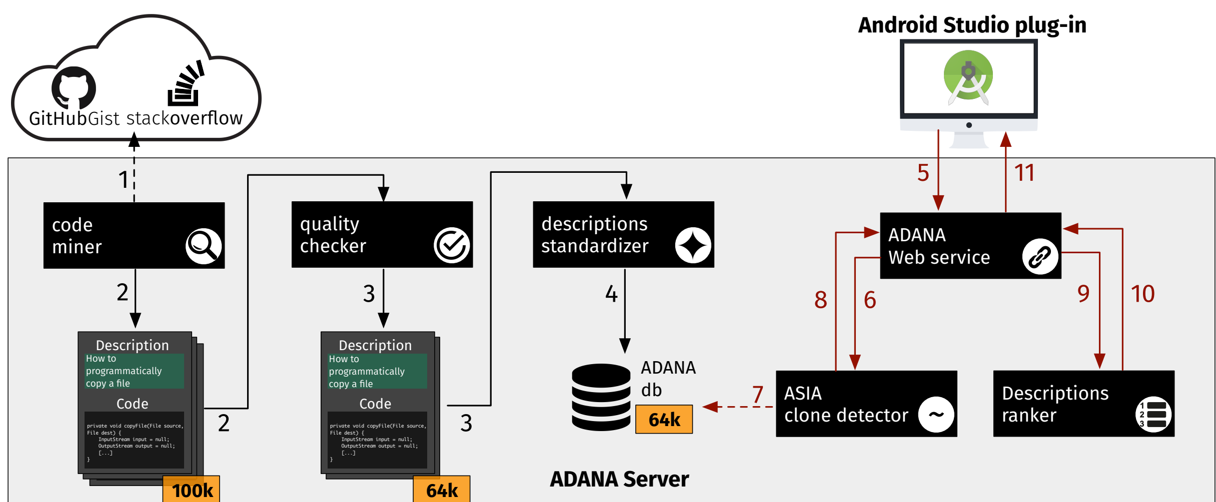


Figure 4.1. The ADANA architecture

ADANA. Once the code clones and their related descriptions are collected, the *descriptions ranker* component selects the best description(s) for the selected piece of code ((9) - (10)). Finally, the selected descriptions are pushed back to the IDE (11) where the developer can integrate them as code comments.

4.2.1 Building the ADANA Knowledge Base

The ADANA knowledge base aims at containing snippets of code accompanied by their description (*e.g.,* "*downloads a file and shows the progress in a ProgressDialog*"). The knowledge base contains pairs of $\langle code, description \rangle$. We instantiated ADANA to the specific problem of documenting Android apps. We populate the knowledge base with code snippets (and related descriptions) relevant for Android apps.

Mining code snippets and related descriptions

ADANA's *code miner* component mines *GitHub Gist* [Gis17] and *Stack Overflow* [Sta17] to identify snippets of code with their related description to populate the knowledge base with pairs of $\langle code, description \rangle$.

Gist. A Gist can be a set of code files, a single code file, or a code snippet. Gists are particularly suited for our approach due to the fact that most of them are accompanied by a short description explaining their purpose (*e.g.,* the `RestartWifi` Gist [res17] is accompanied by the description "*Code snippet to restart wifi interface (Android)*"). To identify relevant $\langle code, description \rangle$ pairs in Gist, we mined from the official Android documentation [and17a] all packages available in the Android APIs (*e.g.,* `android.bluetooth` or `android.support.v4.net`). Then, for each of the mined packages P_i, we used the Gist search feature to identify all code snippets containing an import statement `import` $\mathtt{P_i}.*$, where the $.*$ acts as a wildcard (*i.e.,* it can represent a single class imported as well as the whole package).

We filter out all Gists not written in Java by using the filter provided by GitHub Gist. While the adopted search heuristic does not identify snippets of code not containing any import statement even if they are relevant to Android (false negatives), it is unlikely to select Gists that are not relevant to Android (false positives). In ADANA, we favor quality of data over quantity in the construction of

the knowledge base. This has the drawback of limiting the amount of data available, reducing the number of code snippets that can be automatically documented.

Given a retrieved Gist, we create one knowledge base entry (*i.e.,* one pair $\langle code, description\rangle$) for every Java file composing it (remember that a Gist may have multiple files), using the Java file as the $code$, and the description provided by the user who shared the Gist as the $description$. By using this search heuristic, we extracted 22,864 pairs of $\langle code, description\rangle$ from Gist.

Stack Overflow (SO) is a well-known Q&A website. Besides mining the Q&A posts in SO, we also mined SO documentation, an initiative aimed at creating reference material for developers, collecting code examples showing how to deal with common tasks. Note that the SO documentation was recently shut down. However, we mined it (as detailed in the following) to extract the pairs $\langle code, description\rangle$ needed in the ADANA knowledge base when it was still online (May 2017). Since the extracted pairs still represent a precious source of information, we decided to keep them.

SO documentation is a straightforward resource from where to mine code snippets and their description since it includes pages related to a given topic (*e.g., device display metrics*) showing snippets (and related descriptions) aimed at dealing with common tasks related to the topic (*e.g.,* programmatically capturing the size of the device display).

The *code miner* scrapes all Android-related topics that were already grouped together in the SO documentation. Each topic contains one or more examples (*i.e.,* pairs of $\langle code, description\rangle$), showing how to deal with tasks related to the topic (*e.g.,* the "Intent" topic, contained 19 pairs).

Some preprocessing was needed to identify in each example the related code and description. As for the code, we identify it as the text delimited by the $<$ `pre` $><$ `code` $>$ HTML tags. These tags are used in SO to format the code elements in the questions/answers and, in this case, in the examples reported in the SO documentation.

If multiple Java code snippets were present in the example, we merged them together to create the final code related to the example. Note that since we are focusing on Android, the related SO documentation posts could contain posts mixing code snippets written in Java, C++, and XML as well as makefiles. Since ADANA only supports the automatic documentation of Java code, we used a keywords-based heuristic to identify Java snippets. In particular, given a code snippet, we checked whether any of the regular expressions reported in Table 4.1 matched.

Table 4.1. The heuristic to identify Java code snippets.

#	Regular expression	Language
1	`"^\s*@Override"`	Java
2	`"^\s*<|>\s*"`	XML
3	`"^\s*\w+\s+:="`	Makefile
4	`"^\s*#ifdef|^\s*#ifndef|^\s*#include|^\s*#define |^\s*extern "C"|^\s*public:|^\s*private:|^\s* protected:"`	C++
5	otherwise	Java

The regular expressions were checked in the exact order reported in Table 4.1 and the process was stopped as soon as the first regular expression was matched. The basic idea behind this keywords-based approach is to exploit the unique features of each language (*e.g.,* the `@Override` keyword is only available in Java). A manual analysis of 660 code snippets from our knowledge base (details to follow) confirmed the validity of this simple filtering heuristic, since no non-Java snippets were found.

For the description, we extracted text from the HTML tags having `class` = "`doc` − `example` − `link`" attribute. This text represented a short description of the code shown in the example (*e.g.,* "*Open a*

URL in a browser"), and it is a good fit to concisely document similar snippets of code a developer will select in the Android Studio plug-in. Overall, we extracted 885 $\langle code, description \rangle$ pairs from SO documentation.

Concerning SO Q&A discussions, we mined the SO database dump dated June 2017, retrieving all questions fulfilling the following criteria:

1. *Tagged with a tag containing the word "android"*;
2. *Containing the word "how" in the title*. We use the question title as the description of the code snippets we mine from the answers. A sentence like "How to pass an object from one activity to another on Android" is easily converted into a short description to document a piece of code (*e.g.,* "Passes an object from one activity to another");
3. *Having at least one answer positively rated and/or accepted.* In SO, users can up- or down-vote answers. Also, the person who asked the question can "accept" a specific answer. Since we will use the code snippets reported in the answers as code documented by the question title, we want to make sure that the selected questions have at least one positively judged answer.

Once extracted the set of questions satisfying these constraints, we compose the pairs $\langle code, description \rangle$; the title of each question is used as the description (*e.g.,* "How to disable WiFi in Android"). The descriptions, as for the ones mined from the other repositories, are cleaned and standardized. As for the code, from each accepted/positively rated answer, we extract the code exploiting the < `pre` >< `code` > HTML tags, and use the same keywords-based approach exploited in SO documentation to only consider java code snippets (see Table 4.1). Thus, from a single question we can extract multiple implementations of the same task reported in different accepted/positively rated answers (*e.g.,* different snippets showing how to disable WiFi in Android). We extracted 76,769 $\langle code, description \rangle$ pairs by mining SO Q&A.

The overall mining process resulted in ∼100k $\langle code, description \rangle$ pairs coming from Gist (∼22k), SO Documentation (∼1k), and SO (∼77k) discussions. The larger amount of data coming from SO Q&A discussions is no surprise, considering its popularity.

Table 4.2. Original & standardized descriptions examples

Original Description	**Standardized Description**
How to pass an object from one activity to another on Android - API level 23+	Passes an object from one activity to another
Programmatically download a file with Android, and showing the progress in a ProgressDialog	Downloads a file and shows the progress in a ProgressDialog
A simple wrapper for Scalpel (`https://github.com/JakeWharton/scalpel`) that includes toggle controls accessible from a right-side navigation drawer.	A simple wrapper for Scalpel that includes toggle controls accessible from a right-side navigation drawer

Checking the quality of the mined data

The *quality checker* assesses the suitability of the collected $\langle code, description \rangle$ pairs for the ADANA knowledge base in two steps. The first one aims at removing from the descriptions unnecessary parts. In particular, we made the following changes:

1. New lines are replaced with a space;

2. References to URLs are removed (see *e.g.,* 3rd description in Table 4.2);

3. Common adverbs indicating the need for performing a task programmatically are removed; these adverbs are often present in questions in which developers ask for help on Stack Overflow (*e.g.,* how do I programmatically [...])—see 2nd description in Table 4.2;

4. Expressions clarifying the Java and/or Android context of the task are removed (see 1st description in Table 4.2), since the mined descriptions certainly document Java/Android code thanks to the previous filters.

All the four steps described above are performed by using regular expressions. Table 4.3 reports the regular expressions used for the steps 2), 3), and 4). Note that the regular expression defined for step 4) uses other regular expressions we report in Table 4.4. For example, the first regular expression in Table 4.4, named *semanticVersion*, is used indirectly in the step 4) (last row in Table 4.3) to remove expressions clarifying the Java and/or Android context of the task.

Table 4.3. Regular expressions used by description quality checker for cleaning descriptions

<table>
<tr><th>Step</th><th>Regular expression (python regex)</th><th>Goal</th></tr>
<tr><td>2</td><td>"(?:at|in|on|here|@|see|check|check out|look|look at)?\s*[^a-zA-Z0-9_\]})]?\s*https?://\S+\s*[^a-zA-Z0-9_\[{(]?"</td><td>Removing references to an external resource (e.g., url)</td></tr>
<tr><td>3</td><td>"[(\[]?\s*,?\s*(?:programmatically|dynamically|through code|by code|using code|in code)\s*,?\s*[)\]]?"</td><td>Removing common unnecessary adverbs (e.g., programmatically, through code, using code)</td></tr>
<tr><td>4</td><td>"\s*(?:"+ android_pattern_withParentheses + "|"+ android_pattern_withoutParentheses + "|"+ java_pattern + "|"+ api_pattern_withParentheses + "|"+ api_pattern_withoutParentheses + "|"+ other_pattern + ")\s*,?\s*"</td><td>Removing expressions clarifying the Java and/or Android context of the task</td></tr>
</table>

After the cleaning process, the *quality checker* excludes all $\langle code, description \rangle$ pairs not satisfying a set of three requirements we defined to ensure the quality of the data stored in the ADANA knowledge base. The three requirements have been defined by the first author by manually analyzing a 99%±5% statistically significant sample (computed by using the Student's t-distribution) of the collected data (660 $\langle code, description \rangle$ pairs), looking for possible heuristics to discard low-quality descriptions/code snippets.

This process led to the exclusion from the knowledge base of all $\langle code, description \rangle$ pairs in which:

1. *The description is composed of less than four words.* We aim at removing code snippets accompanied by meaningless/useless descriptions (*e.g.,* "sample", "miniproject", "my application", *etc.*). In the manually analyzed sample (*i.e.,* the 660 instances), we found 110 descriptions having a description composed of less than four words, and only five of them were potentially useful for documenting the related code snippet (*e.g.,* "simple webview"). In the remaining 95.45% of cases, the descriptions were classified as useless to document the code.

Table 4.4. Regular expression used by the ones in Table 4.3

<table>
<tr><th>Pattern name</th><th>Regular expression (python regex)</th><th>Description</th></tr>
<tr><td>semanticVersion</td><td>"\d+(?:[.]\d+)*(?:[.]X|[.][*])*[\b\W]"</td><td>Semantic versioning format, e.g., 4.*, 5.0, or 3.7.x</td></tr>
<tr><td>semanticVersion_widthOptionalParentheses</td><td>"(?:"+semanticVersion+"|[(\[]\s*"+semanticVersion+"\s*[)\]])"</td><td>Semantic versioning with optional parentheses, e.g., (4.*), 5.x.x, or [3.7.x]</td></tr>
<tr><td>semanticVersion_withOptionalRange</td><td>semanticVersion + "(?:\s*\+|\s*and above|\s*and later|\s*and higher|\s*and below|\s*and lower|\s*and up|\s*and further|\s*(?:-|to|and|or)\s*"+ semanticVersion+")?"</td><td>Semantic versioning with optional range, e.g., 3.4+, 4.x and above, 3.0-6.x, or 4.* and 5.*.</td></tr>
<tr><td>android_names</td><td>"(?:pre-)?(?:Cupcake|Donut|Eclair|Froyo|Gingerbread|Honeycomb|Ice Cream Sandwich|IceCream Sandwich|ICS\b|Jelly Bean|JB\b|KitKat|Lollipop|android l\b|Marshmallow|android m\b|Nougat|android n\b|android tv\b)"</td><td>List of Android Code names and their abbreviations, e.g., Nougat, pre-Cupcake, or JB.</td></tr>
<tr><td>prefixWords</td><td>"(?:^|-|\bin|\bon|\bwith|\bfor|\bfrom|\bor|\band|&)"</td><td>A set of common prefix words, e.g., “from” in “from android JB”.</td></tr>
<tr><td>suffixWords</td><td>"(?:beta|versions|version|application|app|devices|device|emulators|emulator|studio)"</td><td>A list of words might appear after referring to an Android version to add more contextual information, e.g., version, application, or emulator</td></tr>
<tr><td>android_fullname</td><td>"(?:android(?!’s)|android\s*"+ semanticVersion_widthOptionalParentheses+"|(?:android)?\s*"+android_names+")\s*"+suffixWords+"?"</td><td>The usual way to refer to a specific Android versions, e.g., Android 3.*, android honeycomb, or android Icecream emulators</td></tr>
<tr><td>android_fullname_withOptioanlRange</td><td>android_fullname + "(?:\s*\+|\s*(?:\band|\bor|/|&|\bto|-|)\s*(?:higher|above|later|upper|up|further|below|lower|low|(?:android)?\s*(?:"+ semanticVersion_widthOptionalParentheses+"|"+ android_names+")))*\s*"+suffixWords+"?"</td><td>Referring to a range of Android versions, e.g., Android 3.x emulators and higher, Android JB or higher devices, or Android 3.x and 4.x</td></tr>
<tr><td>android_pattern_withParentheses</td><td>"[(\[]\s*"+prefixWords+"?\s*(?:"+ android_fullname_withOptioanlRange+"|android)\s*[)\]](?:\s*[|\-:])?"</td><td>e.g., [Android 3.x and 4.x]:, [Android], or (in android JB or higher)</td></tr>
<tr><td>android_pattern_withoutParentheses</td><td>prefixWords+"\s*"+android_fullname_withOptioanlRange+"(?:\s*[|\-:])?"</td><td>Similar to android_pattern_withParentheses, but without parentheses, e.g., in Android ICS:, for android N or higher devices-, or with Android 3.x and 4.x</td></tr>
<tr><td>api_pattern</td><td>"(?:android)?\s*(?:api|sdk)\s*(?:level)?\s*\:?\s*(?:>|<|>=|<=)?"+semanticVersion_withOptionalRange+ "\s*"+suffixWords+"?"</td><td>e.g., Android API level >= 10.x, API level 23+, or Android SDK 21 emulators</td></tr>
<tr><td>api_pattern_withParentheses</td><td>"[(\[]\s*"+prefixWords+"?\s*"+api_pattern+"\s*[)\]]"</td><td>e.g., (from Android API level >= 10.x), [in API level 23+], or (With Android SDK 21 emulators)</td></tr>
<tr><td>api_pattern_withoutParentheses</td><td>prefixWords+"?\s*"+api_pattern+"\s*"</td><td>e.g., from Android API level >= 10.x, in API level 23+, or With Android SDK 21 emulators</td></tr>
<tr><td>java_pattern</td><td>"[(\[]\s*(?:in|for|by|using|with)?\s*java\s*[)\]]"</td><td>e.g., (java), [using java], or (in java)</td></tr>
<tr><td>other_pattern</td><td>"(?:[(\[]\s*"+suffixWords+"\s*[)\]]|/java|with java|at runtime|during runtime)"</td><td>e.g., [emulator], /java, or during runtime</td></tr>
</table>

2. *The description does not contain at least one verb.* Descriptions without verbs are unlikely to represent useful explanations and are too generic to properly document code. In the 660 descriptions in our sample, we found 145 of them do not meet this requirement, and only 19 were classified as potentially useful (*i.e.,* 86.90% of descriptions with no verb were classified as not useful to document the code snippet). Thus, we defined this second heuristic, and use the Stanford CoreNLP toolkit [MSB+14] to identify the presence of verbs.

3. *The code snippet contained less than 50 characters (excluding white spaces) or more than 50 effective code lines (blank lines and comments excluded).* This removes very short and very long code snippets unlikely to represent the implementation of a well defined task. Indeed, in our manually analyzed sample we found 62 "too short" or "too long" snippets, with only 7 classified by the first author as implementing a well defined task. Moreover, not surprisingly, we observed that long snippets usually come with too generic descriptions such as "Custom DigitalClock" which makes them inappropriate for our purpose, *i.e.,* documenting a fine-grained piece of code.

During the manual analysis of the 660 $\langle code, description \rangle$ pairs, we also checked for the presence of non-Java snippets, to verify whether our keyword-based approach to isolate Java snippets works (see Table 4.1). All the 660 inspected snippets were in Java, confirming the validity of the defined approach. After cleaning the dataset by removing all $\langle code, description \rangle$ pairs matching one or more of the three above heuristics, we obtained 63,558 $\langle code, description \rangle$ pairs that represent the ADANA knowledge base.

Standardizing code descriptions

Before adding the $\langle code, description \rangle$ pairs to the ADANA knowledge base, the *description standardizer* converts, using the Stanford CoreNLP toolkit [MSB+14], the mined descriptions in a format suitable to document source code. We defined this process after manually analyzing the previously mentioned sample of 660 descriptions. Table 4.2 reports three example of code descriptions before and after the standardization process.

The *description standardizer* starts by splitting the description into sentences [MSB+14]. Then, it converts all the instances of *how-to* and *howto* (if any) to *how to*. For all sentences starting with *how* (*e.g., how to implement [...], how do you manage [...], etc.*), it: (i) removes the first two words (*e.g., how to*, *how do*, *etc.*) and (ii) removes the 3rd word if it is a personal pronoun (I, you, he, *etc.*). Then, in all sentences the *description standardizer* (iii) converts each infinitive verb not following a modal verb to third person, to give the developer the feeling that the description is referring to "the code" she selects in the IDE (see *e.g.,* the first description in Table 4.2); and (iv) converts each gerund verb following a conjunction to third person (2nd description in Table 4.2). Once the descriptions are standardized, they are stored together with the related code in the ADANA knowledge base (see Figure 4.1).

4.2.2 The ADANA Web Service

ADANA provides a Web service that can be exploited by a REST client, such as the ADANA Android Studio plug-in. The Web service expects from the client an HTTP post request containing a snippet of code. Then, it accesses the knowledge base to look for clones of the provided code snippet, to identify a suitable description provided to the client as an HTTP response. We detail the main steps behind this process as follows (red arrows in Figure 4.1).

The ASIA clone detector

The identification of code clones for the code snippet provided by the client is performed by running our ASIA clone detector on the knowledge base. It is worthwhile to explain why we decided to devise our own clone detector rather than reusing one of those existing in the literature [Kos07, BKA+07, RCK09]. We needed a clone detector able to run on incomplete, uncompilable code. Indeed, most of the code snippets we mined from SO are not complete compilation units. This excludes the use of efficient and well-known tree-based clone detectors such as Deckard [JMSG07]. The obvious choice in these cases is to use text-based clone detectors exploiting Information Retrieval (IR) techniques such as Simian [Har03], that can work on any given piece of code, compilable or not. However, they do not take advantage of the peculiar characteristics of Android code: native Android apps are highly dependent on the Android APIs [MRNAH12, ML13, LVHBCP14]. This can substantially help in identifying whether two snippets of code implement the same feature (*i.e.,* whether they are clones). Indeed, snippets of code implementing the same feature in Android (*e.g.,* identifying the GPS location of the device) are basically "forced" to exploit the same APIs.

For these reasons, we defined ASIA, an approach built on top of standard IR clone detection and tailored for identifying clones in Android-related code. We show in Section 4.3.2 that ASIA achieves a better accuracy as compared to Simian [Har03].

ASIA is designed to detect not only exact clones (type-1 clone), but also clones differing for variable renaming (type-2), for the addition/deletion of few lines of code not changing the main feature implemented in the code (type-3), or even totally different implementations of the same functionality (type-4). Indeed, given the main goal of ADANA (*i.e.,* documenting a piece of code to explain what it implements), any type of code clone represents a valuable source of information.

To explain how ASIA computes the similarity between two code snippets S_i and S_j we introduce two similarity measures. The first is the standard Vector Space Model (VSM) cosine similarity [BYRN99] between the two vectors of words representing S_i and S_j. When applying VSM we (i) normalized the snippets' text using identifier splitting (we also kept original identifiers), (ii) removed English words and reserved programming language keywords, and (iii) used the *tf-idf* weighting schema [BYRN99]. Our replication package [ada19c] provides the list of keywords and the script used in our approach.

The second measure, that we named Android Similarity (*AS*), is a Jaccard similarity [Jac01] between the Android-specific "objects" used in S_i and S_j. We extracted from the Android documentation [and17b] the complete list of Android classes (*e.g.,* `Location`), API methods (*e.g.,* `distanceTo(Location)`), and constants (*e.g.,* `FORMAT_DEGREES`) in the Android framework API. We refer to the set of these Android-specific "objects" as ASO. We compute the AS between the two snippets as:

$$AS(S_i, S_j) = \frac{ASO_{S_i} \cap ASO_{S_j}}{ASO_{S_i} \cup ASO_{S_j}} \tag{4.1}$$

AS represents the percentage of Android-specific objects used by both snippets over the whole set of such objects they use. Given two snippets S_i and S_j, ASIA computes their similarity as:

$$sim(S_i, S_j) = \begin{cases} \alpha \cdot VSM(S_i, S_j) + \beta \cdot AS(S_i, S_j) & \text{if } |ASO| > 0 \\ VSM(S_i, S_j) & \text{otherwise} \end{cases} \tag{4.2}$$

If the two snippets do not contain ASO, their similarity is computed by relying on the VSM, otherwise it is calculated as a weighted sum of their VSM and AS similarity, both defined in [0, 1]. The two weights, α and β, are also both defined in [0, 1] and their sum must be equal to one,

thus ensuring that $sim(S_i, S_j)$ is also in $[0, 1]$. ASIA detects the pair of snippets (S_i, S_j) as clones, if $sim(S_i, S_j) > t$. The tuning of α, β, and t is reported in Section 4.3.2.

Ranking descriptions

Once the list of clones for a code snippet is provided, three scenarios are possible. First, no clones have been found: the client is notified that ADANA is not able to document the snippet of code. Second, only one clone is identified: its description is returned to the client that will use it to document the code. Third, more than one clone is retrieved: ADANA uses the *descriptions ranker* to identify the top clone with the most suitable description for documenting the selected code snippet.

The *descriptions ranker* assigns to the descriptions associated to each code clone a *Quality Score* (QS) indicating their suitability to document a code snippet. The QS for a description D of a clone C ($QS_{D,C}$) is computed by combining together three measures.

The first measure is the similarity (*i.e.,* the sim function in Equation 4.2) computed by ASIA between C (*i.e.,* the code described by D) and the selected code snippet S. The higher the similarity between C and the code snippet S selected in the IDE, the higher the likelihood that D represents a good description for such a snippet.

The second measure is the *Comments Readability* (CR) proposed by Scalabrino *et al.* [SLVPO16]. Extending the Flesch-Kincaid readability index [Fle48], it captures the readability of code comments. We assume that descriptions having a high CR should be preferred over descriptions having a low CR, since the latter might be difficult to comprehend.

The third measure is the *Comments and Identifiers Consistency* (CIC_{syn}), proposed by Scalabrino *et al.* [SLVPO16] to assess code readability. It computes the overlap of terms used in comments (in our case the description) and code identifiers (in our case, the code that we want to document). A high overlap of terms indicates that the comment describes the code well. CIC_{syn} takes into account synonyms (*e.g.,* "display" and "monitor"). CIC_{syn} computation is based on the Jaccard distance of terms used in the description and in the selected snippet.

Since the three measures can be all expressed in $[0, 1]$, given a snippet S_i selected in the IDE, the *descriptions ranker* computes the *Quality Score* ($QS_{D,C}$) for a pair $\langle D, C\rangle$ (*i.e.,* $\langle description, code\rangle$) as:

$$QS_{D,C} = \frac{sim(S_i, C) + CR(D) + CIC_{syn}(D, S_i)}{3} \tag{4.3}$$

Once computed $QS_{D,C}$ for all clones retrieved for the selected snippets, the ADANA Web service returns to the client the description having the highest $QS_{D,C}$ value.

ADANA Android Studio plug-in

Figure 4.2 depicts the ADANA Android Studio plug-in. A developer using ADANA selects a snippet of code she is interested in comprehending, and then invokes ADANA by using the context menu (right click). ADANA requires the developer to select at least three code statements to automatically document (comment) it, to ensure that the ASIA clone detector has sufficient information to reliably identify code clones for the selected snippet.

ADANA shows a *granularity slider* ① to set the granularity of the comments one is interested in retrieving: If the slider is to the left, ADANA looks for clones of the whole code selection and, in case of successful retrieval, only injects a single comment describing the selected code (*i.e.,* a single request is sent to the ADANA Web service). Moving the slider to the right, ADANA decomposes the selected code on the basis of the indentation level, as identified by parsing the AST representing the

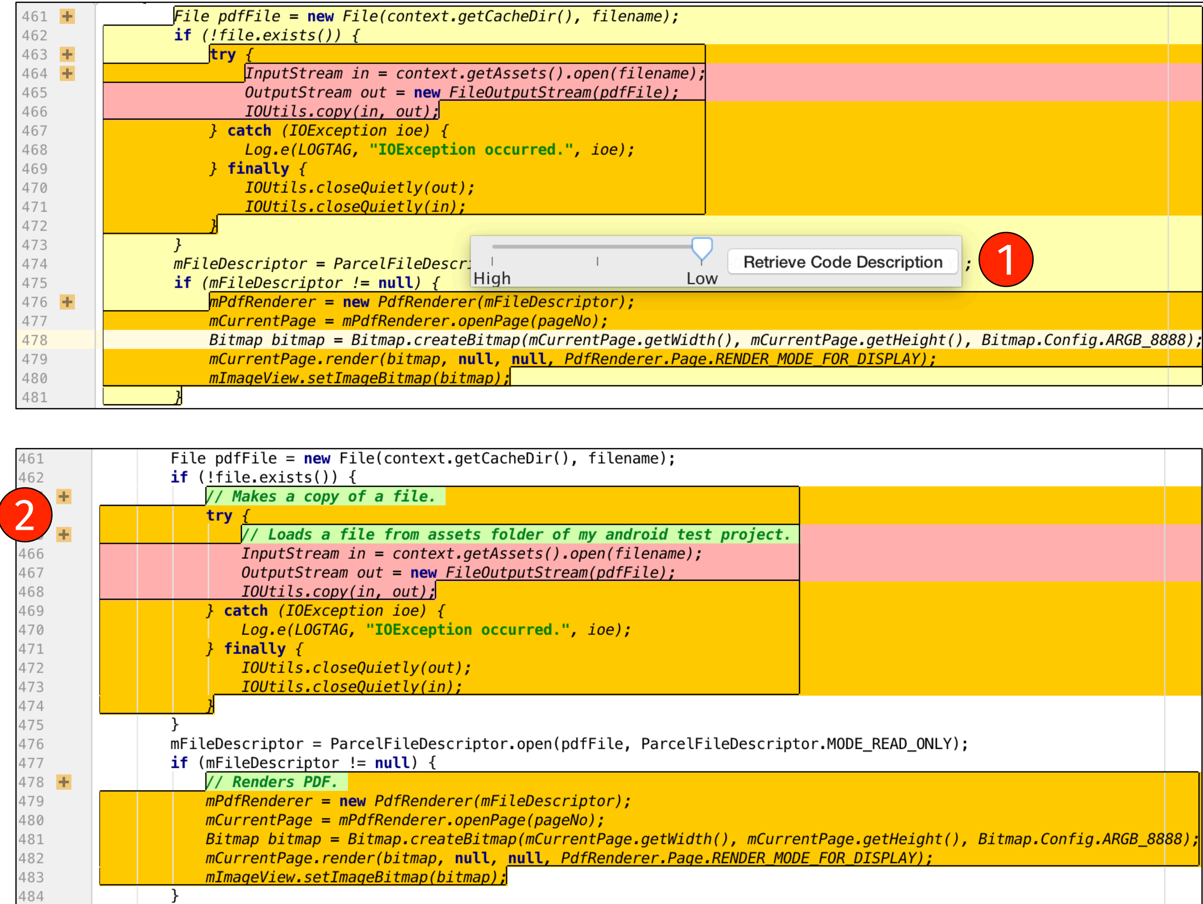


Figure 4.2. ADANA GUI

selection, and looks for clones of (i) the whole code selection, and (ii) the smaller code snippets obtained by decomposing the selection on the basis of the indentation levels. Each of the parts ADANA tries to document is shown in a different color. The maximum value of the *granularity slider* depends on the maximum indentation level of the selected code. Once the developer picks the granularity, she clicks on the “Retrieve Code Description” button close to the slider, obtaining the descriptions retrieved by ADANA for each of the highlighted code portions (see the bottom part of Figure 4.2). By using the code markers added by ADANA in the IDE ②, she can either accept it as is, modify and accept it, or reject it. If she accepts (before/after changing it), both code snippet and the associated comment are added to the ADANA knowledge base.

4.3 Study Design

The study addresses the following research questions (RQs):

RQ$_1$: *What percentage of Android apps’ code can be automatically documented by* ADANA*?*

RQ$_2$: *What is the accuracy of* ASIA *in identifying clones for a given code snippet?*

RQ$_3$: *Does* ADANA *help developers during code comprehension activities performed on Android apps?*

4.3.1 Context Selection and Data Analysis

We describe for each research question its context, the data we collected, and the process adopted to analyze the collected data. The study dataset and the ADANA plug-in are publicly available (see the replication package [ada19c]).

What percentage of Android apps' code can be automatically documented by ADANA?

The focus of RQ_1 is not the correctness/usefulness of the provided comments, but on the *commented code coverage*. We expect the ADANA coverage to improve over time with the increase of data present in its knowledge base.

We selected 16 open-source Android apps from the *open-source-android-apps* [OSA17] GitHub project. The apps were randomly selected from 16 categories from Google Play, ensuring there is one app per category. The list of selected apps is available in Table 4.5. On average, the 16 apps have ∼10k ELOC (*i.e.,* Effective Lines Of Code, excluding blank and comment lines)—min=600, max=37k.

Table 4.5. The 16 apps selected for RQ_1

Category	Selected app (GitHub repository name)	Link to repository
Android TV	XiaoMi/android_tv_metro	https://github.com/XiaoMi/android_tv_metro
Android Wear	romannurik/FORMWatchFace	https://github.com/romannurik/FORMWatchFace
Business	openshopio/openshop.io-android	https://github.com/openshopio/openshop.io-android
Communication	VideoFly/VideoFly	https://github.com/VideoFly/VideoFly
Education	derekcsm/hubble_gallery	https://github.com/derekcsm/hubble_gallery
Finance	nothingmagical/coins-android	https://github.com/nothingmagical/coins-android
Game	snatik/memory-game	https://github.com/snatik/memory-game
Health&Fitness	meghalagrawal/NightSight	https://github.com/meghalagrawal/NightSight
LifeStyle	forezp/banya	https://github.com/forezp/banya
Multi-Media	dkim0419/SoundRecorder	https://github.com/dkim0419/SoundRecorder
News	kinneyyan/36krReader	https://github.com/kinneyyan/36krReader
Personalization	ashutoshgngwr/10-bitClockWidget	https://github.com/ashutoshgngwr/10-bitClockWidget
Productivity	abhijith0505/CarbonContacts	https://github.com/abhijith0505/CarbonContacts
Social Network	Jeffmen/Git.NB	https://github.com/Jeffmen/Git.NB
Tools	cdeange/github-status	https://github.com/cdeange/github-status
Travel	Swati4star/Travel-Mate	https://github.com/project-travel-mate/Travel-Mate

For each app, we simulate a developer selecting snippets of code and invoking the ADANA Web service to document them: given a class C implementing a set of methods M, we use a sliding window of length l to select snippets composed of l contiguous ELOC from the body of each method in M, until all the lines are covered by the sliding window. For example, given a method's body composed of six lines of code and assuming $l = 3$, we automatically extract four snippets of code S_i containing the following lines: S_1={1, 2, 3}, S_2={2, 3, 4}, S_3={3, 4, 5}, S_4={4, 5, 6}. This simulates a developer selecting snippets with three lines of code and asking ADANA to document them. Then, we keep track of the percentage of generated code snippets we were able to document by using ADANA. We experiment with values of l varying between 3 and 21 at steps of 3 (*i.e.,* 3, 6, ..., 21). Our approach does not support selections shorter than three statements. While there is not always a correspondence between ELOC and number of statements, the three ELOC lower-bound ensures valid selections in most of the cases. Note that if a method in the apps considered in our study has less than three statements, we do not consider it.

We are assuming that the developer is not using the *granularity slider* (*i.e.,* she only wants an overall comment for the selected snippet of code). Indeed, given the various granularities we consid-

ered (from 3 to 21 ELOC), simulating the usage of the *granularity slider* is not needed, since the small snippets extracted from a method m (*e.g.,* those composed by 3 or 4 ELOC) are clearly contained into the larger snippets extracted from m (*e.g.,* those composed of 21 ELOCs). We ignore code not present in the method bodies (*e.g.,* import statements) since this is unlikely a real usage scenario for our approach.

To answer RQ_1 we show boxplots of the distribution of the *commented code coverage* obtained in the 16 apps for the considered values of l. Moreover, we present the *commented code coverage* in terms of (i) percentage of ELOC commented, and (ii) percentage of code snippets of length l commented.

What is the accuracy of ASIA in identifying clones for a given code snippet?

We randomly selected from our knowledge base 40 code snippets having between three and twenty ELOCs. We made sure that our approach was able to identify at least one clone for each of the selected snippets, otherwise we replaced it with another code snippet from the knowledge base until all of them met the mentioned requirement.

Then, we asked study participants to assess whether the code clones identified by ASIA for each snippet were true or false positives. The choice of the upper-bound in the snippets' size was driven by the will of considering code snippets that are not too complex and, thus, limit the difficulty and effort required to participants in assessing the correctness of the identified clones. On average, our approach identified 4.8 clones per snippet (min=1, max=9). The set of 40 code snippets and the identified 192 clones are available in our replication package.

The participants were identified by using convenience sampling among the personal contacts of the authors. We invited developers and CS students/professors to take part in our study by using a Web application we developed to perform the following steps. First, we collected demographic data about participants (years of experience in programming, in Java, and in Android, their current position, *etc.*). Each participant was then required to assess the correctness of all clones identified by ASIA for eight snippets randomly selected from the 40 objects of this study. The Web application was designed to automatically balance the number of evaluations for each of the 40 snippets (*i.e.,* the number of participants assessing the correctness of each identified clone was roughly the same).

The eight snippets were presented individually (*i.e.,* each snippet in a different page) to participants, and each clone identified by ASIA for it was shown below the snippet with two radio buttons allowing the participant to express her assessment as: *it is a clone* or *it is not a clone*. We instructed participants to consider all types of clones (*i.e.,* from type-1 to type-4) as valid.

In total, we collected 534 evaluations across the 192 clones of the 40 snippets. We then removed the answers we collected from two participants with zero Android experience, and this resulted in (i) one snippet having no evaluations for its clones, and (ii) one snippet evaluated by only one participant. This latter had 11 clones reported by ASIA for which, only one was not assessed as a true positive. We removed these two snippets and corresponding clones from the analysis to have only clones that were evaluated by at least two participants. Thus, our analysis involves 490 evaluations related to 171 clones of 38 snippets. Each clone was evaluated, on average, by 2.87 participants (median = 2, Q3 = 3).

We analyzed questionnaires completed by 22 participants (11 professional developers, 4 PhD, 3 MSc, and 4 BSc students). Table 4.6 presents demographic information about the participants.

We answer RQ_2 by reporting the percentage of true and false positives classified by the participants[1] as well as by discussing example cases of true and false positives, to highlight strengths and

[1]The percentage of true positives is equivalent to the precision measure (*a.k.a.,* clone detection rate) used by previous papers on clone detection [JMSG07, WTVP16].

Table 4.6. Demographic of study participants (RQ_2)

#years experience in	Average	Median
Programming	7.7	7.0
Java programming	6.1	5.5
Android programming	1.9	1

weaknesses of the ASIA clone detector (Section 4.4.2).

Does ADANA help developers during code comprehension activities performed on Android apps?

We asked 10 professional developers to comprehend a set of snippets of code with and without the help of the comments automatically injected by ADANA. Note that the scenario we aim at simulating in this study is that of developers comprehending code for which comments are not available.

We started by randomly selecting from the 16 mobile apps used in RQ_1 16 methods (one per app) meeting the following criteria:

1. *Having between 10 and 50 ELOC*, to exclude methods that are too trivial or too complex to comprehend.

2. *Having at least one clone identified*, to ensure that ADANA added at least one comment to the methods.

3. *Being self-contained.* One of the authors manually analyzed the selected methods to ensure they were *self-contained*, *i.e.,* they could be comprehended without navigating additional source code (if not those of the Android API framework, available online). This resulted in the replacement of 3 methods.

We ran ADANA on each of the selected methods to have them with and without automatically injected comments. We moved the granularity slider to the right to add as many comments as possible. Also, we removed the original comments (if any) from the methods, to avoid a possible confounding factor and isolating the effect of the injected comments. Also, this is in line with our goal of simulating the real-life scenario in which the developer uses ADANA to understand a method which lacks comments. The comments removal was needed for 4 of the 16 methods (no comments were present in the remaining 12). We refer to the 16 original methods as the *uncommented* dataset, and to the 16 augmented with ADANA comments as the *adana* dataset.

We invited 10 participants via convenience sampling and ran this study via a Web application we developed. Demographic information about the participants are shown in Table 4.7.

Table 4.7. Demographic of participants (RQ_3)

#years experience in	Average	Median	Minimum
Programming	8.4	7.5	1+
Java programming	7.0	6.5	1+
Android programming	2.1	2	1+

Each participant was required to comprehend a subset of eight methods randomly selected from the starting 32. Four of the eight snippets were selected from the *uncommented* dataset, and four

from the *adana* dataset. We made sure that each participant comprehended eight **different** methods (*i.e.,* she did not comprehend twice the same method with and without the ADANA comments). The Web application was in charge of balancing the number of evaluations for each of the 32 methods. We collected 80 evaluations across the 32 methods (2.5 evaluations per method, on average).

The eight methods were presented individually and in a randomized order to mitigate learning and tiring effect. Participants were allowed to browse the Web to collect information about the types, APIs, data structures *etc.*, used in the methods. This was done to simulate the typical understanding process performed by developers. We asked participants to carefully read and fully understand each method. We clarified that "fully understand" can be read as "being able of explaining the method to another developer". Then, they were required to answer three *verification questions* about the method they inspected. The questions were defined, independently, by two of the authors for different sets of methods, with the goal of covering different areas of the method under analysis. This resulted in questions targeting both parts of the method commented and not commented by ADANA, and could represent a confounding factor. We preferred to focus our questions on the whole method rather than only on the parts commented by ADANA to not introduce a strong bias in our evaluation. Of the 48 formulated questions (3 questions × 16 methods), 23 questions explicitly targeted parts of the code commented by ADANA. This also includes *wrong* comments injected by ADANA and not just good comprehension hints, as we discuss in the results section. An example of a comprehended method together with its verification questions is provided in the result section (4.4.3).

The Web app we developed tracked the time needed by each participant to comprehend each of the eight methods and answer the three verification questions. Clearly, this included time spent by participants in browsing the Web looking for information needed to comprehend the snippet. We explicitly asked the participants to not interrupt the comprehension task in order to not introduce a bias in the tracked comprehension time and to report to us in case unexpected interruptions happened. None of the participants reported issues of this type.

To verify if ADANA helps developers in comprehending the code, we used the following two measures for code understandability, defined by Scalabrino *et al.* [SBV+17]:

Actual Understandability (AU). It is computed as the percentage of correct answers the participant provided to the three verification questions. Thus, the metric is defined in [0, 1] range, where 1 indicates high understanding.

Timed Actual Understandability (TAU). It is computed as:

$$TAU = AU\left(1 - \frac{Time}{\max Time}\right) \tag{4.4}$$

where $Time$ is the time needed to comprehend the method and answer the verification questions. The higher the AU is (*i.e.,* the percentage of correct answers), the higher is the TAU; and the higher the $Time$ is (*i.e.,* the time needed to understand the method), the lower is the TAU. Also, TAU is defined in [0, 1]. As done in [SBV+17], we considered the relative time ($\frac{Time}{\max Time}$) so that *TAU* gives the same importance to both the correctness achieved (AU) and the time needed ($Time$).

We computed these two proxies for each of the 80 evaluations by participants. Then, we compare their distributions for methods belonging to the *uncommented* and to the *adana* dataset. A normality check using the Shapiro-Wilk test indicated a statistically significant deviation from normal distribution (p-value< 0.05); hence we use non-parametric statistics. For all tests we consider a significance level $\alpha = 5\%$. We compare the results using the Wilcoxon signed-rank test. Since we do not know *a priori* in which direction the difference should be observed, we use a two-tailed test. We also assess the magnitude of the observed difference using Cliff's delta (d) effect size [GK05], suitable for non-parametric data. Cliff's d ranges in the interval $[-1, 1]$ and is negligible for $|d| < 0.148$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

Note that participants had no information about the purpose of the study and of the fact that comments were injected automatically. We revealed this information at the end of the experiment, and asked to comment their perceived usefulness in an (optional) open question. We report some of the representative comments left by participants.

4.3.2 Experimental Setting

To run our study we have to tune the ADANA's parameters and, in particular, the α, β, and t parameters used by the ASIA clone detector (see Section 4.2.2). To calibrate these parameters we created an oracle reporting true and false positive clones for a set of snippets. We randomly selected eight code snippets from the official Android development guide [ADG17], making sure that the snippets:

- Were implementations of a well-defined task (*e.g.,* activate the WiFi network);
- Were implemented in Java, not involving any usage of files written in other languages (*e.g.,* XML files); and
- Made use of at least one "Android-specific object" (see Section 4.2.2).

The last rule was needed for the tuning of the α and β parameters. To properly set the weights of the VSM and of the *AS* similarities (Equation 4.1), we need to consider cases in which the *AS* can be computed. Then, we took the two longest snippets, and extracted from each of them one "sub-snippet", to simulate the situation in which the developer uses the granularity slider to obtain more fine-grained descriptions of the code. Thus, in total, we included in our oracle ten code snippets.

We identified in the ADANA knowledge base composed of 63,558 $\langle code, description \rangle$ pairs, clones of each selected snippet. In particular, for each snippet we created a set of candidate clones to manually validate by randomly selecting:

1. *Twelve candidate clones from the top-50 results returned by using the VSM similarity*. Thus, 12 clones that are in the top positions when using only textual information to identify clones. Selecting from the top of the ranked list, we expect to increase the likelihood of including true positives in our oracle.
2. *Twelve candidate clones from the top-50 results returned by using the AS similarity*. Thus, 12 clones that are in the top positions on the ranked list when using only information related to Android-specific objects (*i.e.,* classes, API methods, and constants of the Android framework). We made sure to not select in this stage candidate clones that were previously selected when looking at the top-50 results returned by the VSM similarity.
3. *Twelve candidate clones returned in position 51-to-500 by the VSM similarity or by the AS similarity (six for each similarity score)*. We expect these candidates to have a lower likelihood of being true positives, thus allowing us to obtain in our oracle a good mix of true and false positives for each snippet. We decided to not randomly pick from the whole ranked list (we considered up to position 500) to not make the classification of true and false positives trivial (*i.e.,* true positives have a very high similarity value, while false positives have very low similarity values), and thus to ensure a good tuning of the t parameter.

Our oracle includes 10 snippets and 36 candidate clones for each of them. Thus, we performed the tuning on a set of 360 candidate clones that represents a statistically significant sample of the 63,558 snippets present in the ADANA knowledge base with 95%±5% confidence interval (Student's t-distribution). This explains the choice of targeting 36 candidate clones per snippet. Once he obtained the dataset, the first author manually went through all the candidate clones marking each of

them as a true or false positive. The oracle we built is publicly available (see the replication package [ada19c]), and it includes 95 true positives and 265 false positives.

Finally, we run the ASIA clone detector on the built dataset, testing all 220 combinations of α, β, and t obtained varying α and β between $[0, \ldots, 1]$ at steps of 0.1 ensuring $\alpha + \beta = 1.0$ and t between $[0.05, \ldots, 1]$ at steps of 0.05. We evaluate each configuration in terms of (i) its precision, meaning the percentage of correct clones it identifies out of the returned clones, and (ii) its coverage, meaning the percentage of snippets for which it is able to identify at least one correct clone. Before discussing the results some clarifications are needed. First, we did not use recall, since for our specific application (*i.e.,* automatically identifying a description for a given code snippet) what we really care is to find at least one suitable description. This is why we rely on the snippets' coverage. Second, ADANA is the classic application in which precision is **much** more important than recall. Indeed, the scenario in which our tool will be used is that of a developer experiencing difficulties in comprehending a piece of code and, thus, asking for help to ADANA. In such a scenario it is better to just report to the developer a void result (*i.e.,* "*I am not able to document this code*") rather than providing a wrong description confounding the developer even more.

Figure 4.3 shows the results of the tuning process for (i) VSM similarity (*i.e.,* $\alpha = 1$ and $\beta = 0$), red line and bars (ii) *AS* similarity (*i.e.,* $\alpha = 0$ and $\beta = 1$), blue line and bars, and (iii) the best combination (in terms of high precision, good coverage compromise) of the two (*i.e.,* $\alpha = 0.5$ and $\beta = 0.5$) we identified. The results are shown for precision (y-axis, lines) and coverage (y-axis, bar chart) when varying the t threshold (x-axis). The results for all experimented combinations are available in our replication package.

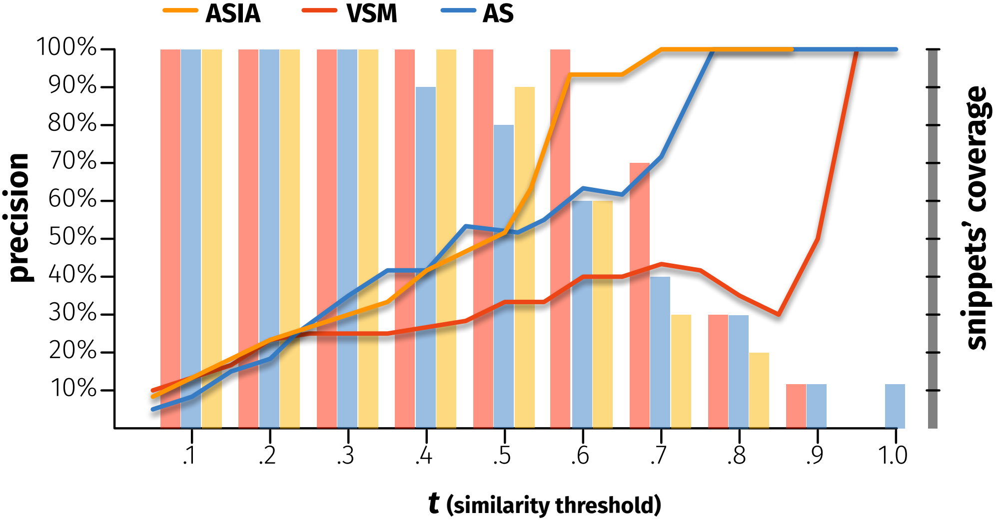


Figure 4.3. Tuning of the α, β, and t ASIA parameters

The VSM exhibits the lowest precision, with good values (100%) exhibited only when the coverage drops at 10%, meaning that it is able to identify true positives code clones for only one of the ten snippets in our oracle. The Android Similarity (*AS*) performs better, while still obtaining very low coverage (30%) when the precision values become acceptable (> 75%). Finally, the best combination we identified for the ASIA clone detector is able to reach very high values of precision (93%) when the coverage still exhibits a good 60% (*i.e.,* we should be able to automatically document six out of ten snippets with such a level of precision) by using $t = 0.65$. $\alpha = 0.5$, $\beta = 0.5$, and $t = 0.65$ is the default ADANA configuration, and the one we will use in our study.

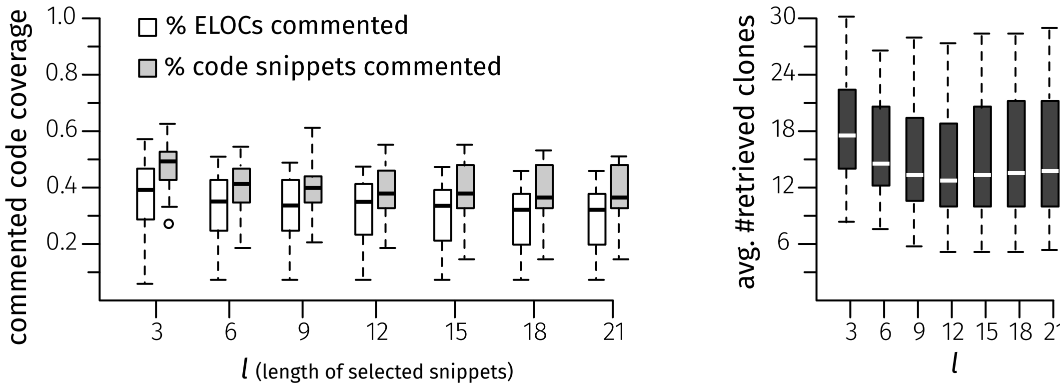


Figure 4.4. Commented code coverage (left) and average number of retrieved clones (right)

On the performance of the Simian [Har03] clone detector. As explained in Section 4.2.2, the obvious alternative to using ASIA in ADANA would have been to exploit a clone detector based on IR techniques. This is due to the need for running the clone detection on incomplete, uncompilable code. For this reason, before finalizing our decision of using ASIA, we also ran the Simian clone detector on the same dataset we used for the ASIA's tuning. Simian does not have a "similarity threshold" to tune, but simply returns the set of clones it is able to identify in a given dataset. However, it has a number of parameters to set (see the replication package [ada19c] for the complete configuration we used). We set those parameters to make sure that Simian did not only look for exact (type-1) clones (*e.g.,* we ignore differences related to the name of the variables, constant values *etc.*). Simian was able to identify a correct clone for 50% of the 10 snippets (*i.e.,* coverage=50%) with a precision of 77%. ASIA, in the configuration we adopted, achieves a higher coverage (60%) accompanied by a higher precision (93%).

4.4 Results & Discussion

In the following sections we answer the three research questions formulated in Section 4.3.

4.4.1 What percentage of Android apps' code can be automatically documented by ADANA?

Figure 4.4 reports the *commented code coverage* achieved by ADANA on the 16 subject apps. The white box plots show the percentage of ELOC that were automatically documented by ADANA, while the grey ones report the percentage of automatically selected code snippets for which the ASIA clone detector was able to identify at least one clone. The results are shown when varying the length of the selected code snippet (l) between 3 and 21 at steps of 3 (x-axis). Finally, the black box plots show the average number of clones retrieved by ADANA for the snippets of code it was able to document. In total, this evaluation involved 114,499 code snippets of different length.

Looking at Figure 4.4, the first noteworthy finding is the stable coverage trend when varying the length of the selected snippets. Indeed, the median percentage of documented ELOC varies between 30% and 36%, while the percentage of documented code snippets between 38% and 45%. On the negative side, this indicates that ADANA generally cannot help developers in comprehending almost two-thirds of the apps' source code. While this might look like a negative result, it is worth remembering that the approach is fully automated and it is unrealistic to expect much higher coverage. Also, these percentages represent a lower-bound that is likely to increase over time with the growth

in the size of the ADANA knowledge base as more *Stack Overflow* discussions and *GitHub* Gists are posted.

Similar observations can be made for the average number of clones retrieved by ADANA for snippets it was able to document, with the median ranging between 14 and 18, indicating that ADANA is often able to retrieve several descriptions for a given snippet. This justifies the need for the *descriptions ranker* in our approach.

The app on which ADANA achieves the lowest coverage (∼7% ELOC and ∼12% snippets coverage) is FORM Watch Face for Android Wear [for17]. It is developed for Android wearable devices (*i.e.,* watches) and, in the ADANA knowledge base, we only have 54 $\langle code, description \rangle$ pairs containing the word "*wear*": ADANA does not have enough relevant entries in its knowledge base. The code ADANA is able to document for this app is mostly standard Android code that can also be found in mobile phone apps.

10-bitClockWidget [10b17] is instead the app for which ADANA achieves highest coverage levels (∼48% ELOC and ∼50% snippets coverage). It implements a clock widget that can be embedded in the home screen. Android widgets strongly rely on the classes implemented in the `android.appwidget` package of the Android API framework, that does only contain five classes, thus promoting the use of similar code across different widgets.

Summarizing, the achieved results show a good coverage level exhibited by ADANA on the 16 subject apps, with almost one-third of the apps ELOC that could be automatically documented. Clearly, we did not focus on the correctness of the identified clones and, as a consequence, on the usefulness of the retrieved documentation, which is the object of the next research questions.

4.4.2 What is the accuracy of ASIA in identifying clones for a given code snippet?

As previously said, we collected 490 clones evaluations related to 171 clones of 38 code snippets. 381 of the 490 evaluations were marked as true positive, which accounts for a *percentage of true positives (a.k.a., precision)* of 77,76%.

Concerning the participants' agreement, for 118 clones out of 171 at least two-thirds of the evaluators classified the candidate clone as a true positive, while for 18 cases at least two-thirds of the evaluators agreed on classifying the candidate as a false positive. This means that moving the focus on the single code clones, by adopting a majority-voting schema (*i.e.,* by considering a clone as a true positive only if the majority of the evaluators classify it as a true positive), we obtain a precision of 69.00% (118/171). In more detail, for 99 out of 171 clones, all the participants evaluating the candidate clone agreed with ASIA (*i.e.,* answered "Yes, it is a clone"), while in 14 cases all the evaluators disagreed with our approach (*i.e.,* answered "No, it is not a clone").

We also investigated the types of code clones detected by ASIA. The first author manually analyzed the 118 clones classified by the majority of participants as true positives with the goal of classifying each of them as a type-1, type-2, type-3, or type-4 clone. We found no instances that can be considered as type-1 clones, four type-2 clones, 102 type-3 clones, and 12 type-4 clones. Thus, most of the clones identified by ASIA differ for the addition/deletion of few lines of code not changing the main feature implemented in the given code snippet.

Listing 4.2 reports an example of a type-3 clone detected by ASIA for the code snippet shown in Listing 4.1 classified by all participants as a true positive. The snippet and corresponding clone create a `Bitmap` object from a `URL`; note that the clone implements the same feature of the code snippet, but it has additional statements and there are changes in the identifiers.

```
URL url = new URL("http://www.yourdomain/your/path/image.jpg");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setDoInput(true);
connection.connect();
final InputStream input = connection.getInputStream();
Bitmap yourpic = BitmapFactory.decodeStream(input);
```

Listing 4.1. Code snippet creating a bitmap object from a URL

```
public Bitmap getBitmapfromUrl(String imageUrl)
{ try {
      URL url = new URL(imageUrl);
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setDoInput(true);
      connection.connect();
      InputStream input = connection.getInputStream();
      Bitmap bitmap = BitmapFactory.decodeStream(input);
      return bitmap;
   } catch (Exception e) {
      e.printStackTrace();
      return null;
   } };
```

Listing 4.2. Detected (true positive) clone for Listing 4.1

Conversely, Listing 4.4 depicts an example of a clone detected by ADANA for the snippet in Listing 4.3, classified by all the evaluators as false positive.

```
btn.setTag(textView.getText().toString());
btn.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      // TODO Auto-generated method stub
      String s =(String)v.getTag();
   } });
```

Listing 4.3. Code snippet declaring a listener for a button

```
textViewField.setOnLongClickListener(new OnLongClickListener() {
  @Override
      public boolean onLongClick(View v) {
         // TODO Auto-generated method stub
         return false;
      } });
```

Listing 4.4. Detected (false positive) clone for Listing 4.3

The snippet in Listing 4.3 (i) declares a click listener for a button and the corresponding `onClick` method, and (ii) shows how to set and get the button tag. Instead, the clone (Listing 4.4) implements a long click listener for a text view. This false positive is due to the extremely high VSM between the candidate clone and the snippet (*i.e.,* 0.79). Such a high value is due to several shared terms between the two snippets (*e.g.,* click, listener, text, view, auto, generated *etc.*). Some of these terms are due to the comment automatically generated by the IDE (*i.e.,* TODO [...]). These comments should be removed by matching them with regular expressions before computing the clones' similarity. This is something we implemented in ADANA after the results of this study and thanks to this example.

Another example of candidate clone classified by all participants as false positive is related to a

code snippet which converts pixels to DIPs (Density Independent Pixels) for which ASIA identified a clone doing the opposite (*i.e.,* converting DIPs to pixels). Also, the identification of this false positive clone is partially due to the high VSM similarity between the candidate clone and the snippet, due to several terms shared between the two methods, as well as to the co-usage of Android constants such as `DisplayMetrics.DENSITY_DEFAULT`, needed in both methods.

Finally, for 35 clones the participants did not reach an agreement (*i.e.,* the answers were distributed equally towards true and false positives). For example, we had a snippet showing how to create a context menu and add items to it and a clone identified by ASIA for it implementing the same feature. However, in the snippet the items were statically added using String variables (*e.g.,* `menu.add`(“Option1”)), while in the clone the menu items were added dynamically according to an item selected on a list view. While at high-level the snippet and the clone implement the same feature (*i.e.,* create a context menu), the implementation differs in how the menu is populated.

Participants did not reach an agreement also for the code shown in Listing 4.6, reporting a clone detected by ASIA for Listing 4.5. While the two code snippets focus on very similar tasks, *i.e.,* writing/reading into/from `SharedPreferences`, the exact actions they perform on the `SharedPreferences` object are different. Three out of the six participants involved in the assessment of this clone marked it as a false positive.

```
SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(YourActivityName.this);
Editor edit1 = remembermepref.edit();
edit1.putInt(totalbalance_key,totalBalance);
edit1.commit();;
SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(YourActivityName.this);
int totalbalance = pref.getInt(totalbalance_key);
```

Listing 4.5. Code snippet writing/reading an int value into/from shared SharedPreferences

```
PreferenceManager.getDefaultSharedPreferences(this).edit().putInt(your_key, <Your_value>).commit()
    ;;
PreferenceManager.getDefaultSharedPreferences(this).getInt(your_key, <Default_value>);
```

Listing 4.6. Clone detected for Listing 4.5

Note that in this study we did not consider the false negatives (*i.e.,* clones of the considered code snippets that were present in our knowledge base but were not retrieved by ADANA) and, as a consequence, the ADANA’s recall for two reasons. First, given a code snippet, it is practically impossible to manually identify all its clones in a database of 64k snippets. Thus, without having a complete oracle, it is not possible to identify false negatives. Second, considering the goal of our approach (*i.e.,* identifying clones to “reuse” code descriptions), what we care about is that the clones identified by ADANA are true positives, to avoid the injection of wrong comments in the code to document.

In summary, ADANA is able to detect clones for Android code snippets with a precision of $\sim 70\%$, by relying on lightweight textual analysis. Further work should be devoted to improving the precision of ADANA with static analysis techniques that might resolve the issues in the aforementioned examples.

4.4.3 Does ADANA help developers during code comprehension activities performed on Android apps?

Figure 4.5 reports the understandability (*i.e.,* correctness achieved in the verification questions—left side) and the TAU (*i.e.,* Timed Actual Understandability, taking both correctness and time needed for

the comprehension into account—right side) achieved by developers when comprehending methods commented (*adana* group) and not commented (*uncommented* group) by our approach.

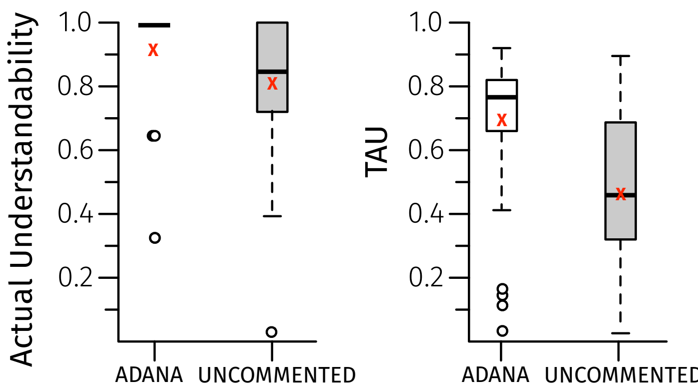


Figure 4.5. Participants' understandability for methods commented (ADANA) and not (*uncommented*) by ADANA. The red x represents the average of distribution, while the circles show the outliers.

Participants obtained a better understanding of the method under analysis when comments injected by ADANA were present. Indeed, the average understandability was 0.92 (median = 1.00) in the *adana* group, and 0.77 (median = 0.83) in the *uncommented* group. This difference is statistically significant (p-value=0.03) with a medium effect size (d=-0.33).

The difference is even more marked when assessing the comprehension level by also considering the time participants spent understanding the methods and answering the verification questions. The average TAU is 0.66 in the *adana* group (median=0.75), and 0.45 in the *uncommented* group (median=0.44). Such a strong difference is due in part to the higher understandability achieved by participants thanks to the comments injected by ADANA, but mostly to the time developers saved in comprehending the methods when working in the *adana* group. Indeed, on average, participants spent 99 seconds (median=87) per method when comments by ADANA were present as compared to 140 (median=126) when they were not present. Also, in this case, the difference is statistically significant (p-value<0.01) but with a large effect size (d=-0.48).

While ∼100 seconds looks insufficient to comprehend a method, it is worth remembering that we also had in our sample methods composed of only 10 ELOC. Also, the participants were all professional developers with Android experience, and we asked them to perform the comprehension activity in the shortest time possible to make sure they did not stop while performing a task (thus, introducing bias in the collected data). Given the maximum comprehension time we registered for a single method (*i.e.,* 318 seconds), we are confident that the developers did their best to understand the code and answer our questions in the shortest time possible.

```
public void onTextChanged(CharSequence s, int start, int before, int count){
  // Shows a Clear button after the first character pressed and hides it when the text is empty
  if(s.length() > 0){
    clear_button.setVisibility(View.VISIBLE);
    searchText = s.toString();
  }
}
```

Listing 4.7. Example of useful injected comment

Listing 4.7 shows (part of) one of the methods the developers were asked to understand. The comment in the method is automatically injected by ADANA and helped the participants in quickly

understanding under which circumstances the clear button is visible on the screen (*When is the clear_button shown to the user?* was the first question we asked about this code snippet). While developers were able to fully comprehend and correctly answer all verification questions for this code snippet both with and without the injected comments, they saved, on average, two minutes of comprehension activity thanks to the ADANA injected comments.

```
// Resizes image before decoding it to bitmaps.
float widthRatio = ((float) rotatedWidth) / ((float) MAX_IMAGE_DIMENSION);
float heightRatio = ((float) rotatedHeight) / ((float) MAX_IMAGE_DIMENSION);
float maxRatio = Math.max(widthRatio, heightRatio);
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = (int) maxRatio;
srcBitmap = BitmapFactory.decodeStream(is, null, options);
[...]
// Rotates a bitmap.
Matrix matrix = new Matrix();
matrix.postRotate(orientation);
srcBitmap = Bitmap.createBitmap(srcBitmap, 0, 0, srcBitmap.getWidth(), srcBitmap.getHeight(),
    matrix, true);
```

Listing 4.8. Example of useful injected comment

Listing 4.8 reports another example of correct and useful comments injected by ADANA. In particular, Listing 4.8 shows two parts of a method subject of our study for which ADANA injected two comments (*i.e.,* "`//Resizes image before decoding it to bitmaps`" and "`//Rotates a bitmap`") both correctly describing the method behavior, helping participants to increase their average actual understandability (AU) by 22%, while still saving, on average, one minute of comprehension activity.

Interesting insights were also provided by participants when answering to the last open question in which we revealed that the comments were automatically injected and asked participants to provide their thoughts. One of the participants wrote "*comments were useful to comprehend at least parts of the snippets*", confirming the potential usefulness of ADANA. Another one said "*I noticed one case in which the comment was partially wrong, since it referred to loading JSON from a webpage, while the method was parsing the webpage but not as a JSON, other comments were good*". The developer is referring to a method in which ADANA injected this comment "`//JSON parser from web page`" in a method that stored the output of an HTTP request into a String for further analysis.

This clearly represents a case of "false positive" comment. We did not observe any strong impact of this comment on the participants' performance, likely due to the fact that it was compensated by the still useful context hint (*i.e.,* parsing a web page). A case in which the participants performed equally both with and without the comments injected by ADANA is represented by code snippet #8 (see replication package). This case is interesting since, despite the fact that the injected comment ("`//Reads Distinct Contacts with Contact Number and Names`") correctly describes the snippet, it did not benefit the correctness achieved by participants nor the time they spent comprehending the code. Our conjecture is that the complexity of the code snippet, including two `while` loops, three `if` statements, and one `switch-case` statement, probably pushed the developers to carefully inspect the whole code in both scenarios, thus reaching a similar comprehension level in roughly the same amount of time (∼200 seconds) when working with the two treatments.

Altogether, ADANA seems to help developers in code comprehension activities performed on originally *uncommented* code snippets. This especially results in time saved for the code comprehension.

4.5 Threats to Validity

Construct validity. In RQ_1 we mimic developers selecting snippets of code to assess the ADANA code coverage. While we experimented with code snippets of different length, our simulation might not be realistic of typical snippets selected by developers. Also, we considered the "coverage" of all apps' ELOC as equally important, which is questionable (*e.g.,* the code implementing the application logic is likely the one most important to document).

Internal validity. To avoid bias in the experiments performed to answer RQ_2 and RQ_3, we made sure that participants were neither aware of the investigated research questions nor of the general goal of the tasks we required them to perform. Also, we decided not to include in our studies participants without Android experience to have a more homogeneous population and to avoid a strong influence of the participants' knowledge/skills as a confounding factor. Finally, we made sure to have multiple evaluators for each candidate clone (RQ_2) and for each method participants had to comprehend (RQ_3).

The three requirements used in the *quality checker* to exclude $\langle code, description \rangle$ pairs likely having a low quality have been defined by the first author, thus introducing possibly subjectivity bias. However, the requirements he defined have been discussed among all authors, also looking at the pairs discarded thanks to their application.

Also, in RQ_3 we limit our analysis to methods having a size between 10 and 50 ELOC, with the goal of excluding from our study methods that were too trivial or too complex to understand. However, these thresholds have been defined based on the authors' development experience, and experimenting with methods of different size could lead to different findings.

Conclusion validity. We address threats to conclusion validity by using appropriate statistical tests and effect size measures to support our claims. While we observed a positive effect of ADANA on code comprehension activities performed on *uncommented* code snippets, we are not claiming its usefulness in a scenario in which the code is commented, since a different study design would be needed to assess this.

External validity. The generality of our results is bounded by the limited number of apps (RQ_1), snippets (RQ_2), and methods (RQ_3) used in our study, as well as by the number of participants (RQ_2 and RQ_3). For example, it is possible that the coverage level observed on the apps selected for RQ_1 does not generalize to other apps.

4.6 Summary and Conclusion

We presented ADANA, an online-resources-mining approach and a tool to collect $\langle code, description \rangle$ pairs that can be reused to automatically document similar pieces of code — given a target snippet to inspect —, which are identified by using the ASIA clone detector we devised. ADANA is currently tailored to work on Android apps but could be adapted/extended to support the documentation of any software system. For example, assuming the will to extend the ADANA support to C++ systems, this would mostly require extensions to the (i) clone detector, by adding a detector designed to work on C++ code, and (ii) knowledge base, mining C++ code snippets.

While the results achieved in the performed evaluation are already encouraging, we believe that the strength of ADANA lies in the always increasing amount of data that it will be able to exploit in the mined online resources, making ADANA better and better over time. We further discuss our future plan in Chapter 7.

5

Software Documentation Issues Unveiled

SOFTWARE DOCUMENTATION provides developers and users with a description of what a software system does, how it operates, and how it should be used. For example, technical documentation (*e.g.,* an API reference guide) aids developers during evolution/maintenance activities, while a user manual explains how users are to interact with a system. Despite its intrinsic value, the creation and the maintenance of documentation is often neglected, negatively impacting its quality and usefulness, ultimately leading to a generally unfavorable take on documentation.

Previous studies investigating documentation issues have been based on surveying developers, which naturally leads to a somewhat biased view of problems affecting documentation. We present a large scale empirical study, where we mined, analyzed, and categorized 878 documentation-related artifacts stemming from four different sources, namely mailing lists, Stack Overflow discussions, issue repositories, and pull requests. The result is a detailed taxonomy of documentation issues from which we infer a series of actionable proposals both for researchers and practitioners.

Structure of the Chapter

- **Section 5.1** provides the motivation for this chapter.
- **Section 5.2** describes the study design, and introduces our central research question, which is addressed in **Section 5.3**, where we present a taxonomy of 162 types of documentation issues accompanied by in-depth discussion of it.
- **Section 5.4** presents the threats that could affect the validity of our findings.
- Finally, **Section 5.5** concludes this chapter.

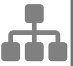
Supplementary Material

All the data used in this chapter is publicly available at our replication package [doc19]. Specifically, our replication package includes:

- The list of keywords used in the automatic selection of the documentation-related artifacts
- The predefined and final lists of labels of our manual analysis
- The database of labeled sentences for each artifact
- The high-resolution version of Figure 5.1 annotated with additional details

Accomplishments in a Nutshell

Software Documentation Issues Unveiled [ANVM+19]
Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza
In *Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*, pp. 1199–1210. IEEE, 2019.



Taxonomy of Documentation Issue Types
A *comprehensive taxonomy consisting of 162 types of documentation issues* linked to (i) the information it contains, (ii) how the information is presented, (iii) the documentation process and (iv) documentation tool support (See Figure 5.1).

In-Depth Discussion of Documentation Issues and Implications For each category of documentation issues type, we present interesting examples, common solutions, and discuss their implications in software research and practice, deriving a series of actionable proposals both for researchers and practitioners.

Artifact Labeling Web app
A Web app that we developed to support artifact labeling. Our platform enables us to label artifacts at sub-sentence level and to resolve the conflicts among taggers.

5.1 Introduction

Good ~~old~~ documentation, the ideal companion of any software system, is intended to provide stakeholders with useful knowledge about the system and related processes. Depending on the target audience, the contents of documentation varies. For example, technical documentation (*e.g.,* API reference guides) describes information about the design, code, interfaces and functionality of software to support developers in their tasks, while user documentation (*e.g.,* user manuals) explains to end-users how they should use the software application.

Despite the undeniable practical benefits of documentation during software development and evolution activities [FL02, LSF03b, KM05, ZGYS+15], its creation and maintenance have been often neglected [FL02, FWG07, CH09, LVLVP15, KM05, ZGYS+15], leading to inadequate and even inexistent documentation. These and other aspects of documentation (*e.g.,* needs, learning obstacles) have been investigated through interviews with and surveys of practitioners, with the general goal of identifying the root causes of documentation issues (*e.g.,* inaccuracy, outdatedness). To address these issues (at least partially), different approaches and tools have been proposed to aid developers during software documentation, including its automatic generation [FL02, MM16]. For example, a recent proposal by Robillard *et al.* [RMT+17] suggests a paradigm shift towards systems that automatically generate documentation in response to a developer's query, while considering her working context.

However, to achieve high-quality automatic documentation systems, we require first a deep understanding of software practitioners' needs. Although existing studies have revealed some of these needs, their results are limited by the low number and lack of diversity of practitioners questioned and documentation artifacts analyzed.

To overcome these limitations, we qualitatively analyzed different types of artifacts from diverse data sources and identified the *issues that developers face when dealing with documentation*. Specifically, we mined open-source software repositories and examined a set of 878 artifacts corresponding to development emails, programming forum discussions, issues and pull requests related to software documentation. For each artifact, we determined the reported issue, the type of documentation presenting it, and the proposed solution, as well as the documentation tools discussed. Based on our analysis, we built a *comprehensive taxonomy consisting of 162 types of documentation issues* linked to (i) the information it contains, (ii) how the information is presented, (iii) the documentation process and (iv) documentation tool support. We describe and exemplify each category of documentation issues. We also discuss their implications in software research and practice, deriving actionable items needed to address them.

5.2 Study Design

Our goal is to answer the following research question (RQ):

> *What are the documentation issues faced by developers?*

5.2.1 Data Collection

Our data collection consists of two steps. First, we adopt an automatic process based on keyword matching to mine candidate artifacts related to documentation from the four analyzed sources (*i.e.,* emails, issues and pull requests of open-source projects, and Stack Overflow threads). Then, we manually analyze a statistically significant sample of artifacts to categorize them based on the issues they discuss, the solutions they propose, and the type of documentation they involve.

Identification of Candidate Artifacts Related to Documentation Issues

Table 5.1 summarizes the artifacts automatically collected from the four sources (see column "candidate artifacts"). We discuss the process adopted in each case.

Table 5.1. Study Dataset

Source	Candidate Artifacts	Manually Analyzed	False Posit.	Valid Artifacts	Labeled Sentences
Issues	394,504	345	19	324	562
Mailing Lists	6,898	101	5	95	220
Pull Requests	375,745	332	21	310	581
Stack Overflow	28,792	100	4	95	185
Overall	**805,939**	**878**	**49**	**824**	**1,548**

Stack Overflow (SO). We mined from the official SO dump of June 2018 all discussions having a question labeled with a documentation-related tag. To determine these tags, we searched for all tags related to documentation and documentation tools in the SO tag page by using the keywords *doc*, *documentation* and *documentor*. The latter term is known to be part of the name of tools supporting software documentation. One author then inspected all the tags resulting from these three searches to identify the ones actually related to software documentation and/or documentation tools. During the inspection, the author read the tag name, the tag description and some of the questions in which the tag was used. This process resulted in the selection of 23 tags (*e.g.,* code-documentation, php-documentor, design-documents) that were used to search for the related discussions in SO. The first 30 results (discussions) returned by the 23 searches were manually inspected to look for additional documentation-related tags missed in the first step. The process was iterated with the newly founded tags until no new tags were found in the top 30 results of the tag searches. This resulted in a total of 78 (23+55) documentation-related tags (available in our replication package).

Next, we queried the SO dump to extract all discussions having a question with a non-negative score and tagged with one or more of the relevant 78 tags. We removed questions with a negative score to filter out irrelevant discussions. This process resulted in the selection of 28,792 discussions. For each of them, we kept the question, the two top-scored answers and the accepted answer (if any).

GitHub Issues and Pull Requests. We downloaded the GitHub Archive [Gri] containing every public GitHub event occurring between January 2015 and April 2018. While older data is available, we excluded it since some of the information needed for our study was only archived starting from 2015. We extracted all events of type *IssuesEvent*, *IssueCommentEvent*, *PullRequestEvent* and *PullRequestReviewCommentEvent*.

These events capture the opening/closing of issues and pull requests as well as all the discussion held for them through comments. A detailed description of these event types is available online [Git]. Then, we selected issues and pull requests from projects having at least ten forks and/or stars to exclude "toy" projects. Finally, we adopted a keyword-matching approach to extract issues and pull requests related to documentation. We started from the 78 SO tags previously mentioned and *converted* them into 56 "keywords". This means, for example, that the SO tag *design-documents* was converted into *design doc* (to match "design document", "design documents" and "design doc"), while tags including the word "documentation" (*e.g., xml-documentation*) were replaced with the keyword *documentation*, since matching this keyword will also match the more specific ones. We also added 11 keywords that we considered relevant but were not derived from any of the 78 SO

tags. For example, while the keyword *api doc* was derived from the SO tag *api-doc*, we also added *api manual*. In total, we defined 66 documentation-related keywords. The complete list of these keywords is available in our replication package [doc19].

We extracted all the issues/pull requests having at least one of the 66 keywords in their title and/or in their first post (*i.e.,* the one opening the issue or the pull request). This resulted in the selection of 394,504 issues and 375,745 pull requests.

Mailing Lists. We built a crawler to mine the mail archives of the Apache Software Foundation (ASF) [Fou]. The ASF archives all emails exchanged in the mailing lists of the projects it runs. Each of its 295 projects has several mailing lists focused on different topics. We mined all mailing lists named *docs* (discussions related to documentation), *dev* (discussions among developers) and *users* (discussions involving both users and developers), for a total of 480 mailing lists. For the threads extracted from the *docs* mailing lists, we did not apply any filter. For the threads extracted from the *dev* and the *users* mailing lists, we only selected those containing in the subject at least one of 66 documentation-related keywords we previously defined. This resulted in the extraction of 6,898 email threads, each one composed by one or more messages.

Manual Classification of Documentation Issues

Once we collected the candidate artifacts, we manually analyzed a statistically significant sample ensuring a 99% confidence level ± 5%. This resulted in the selection of 665 artifacts for our manual analysis, out of the 805,939 artifacts collected from the four sources. Since the number of collected artifacts is substantially different between the four sources (Table 5.1), we decided to randomly select the 665 artifacts by considering these proportions. A simple proportional selection would basically discard SO and mailing lists from our study, since issues and pull requests account for over 90% of our dataset. Indeed, this would result in the selection of 311 pull requests, 326 issues, 24 SO discussions and 6 mailing list threads. For this reason, we adopted the following sampling procedure: for SO and mailing lists, we targeted the analysis of 96 artifacts each, ensuring a 95% confidence level ± 10% within those two sources. For issues and pull requests, we adopted the proportional selection as explained above. This resulted in 829 artifacts to be manually analyzed (99% confidence ± 4.5%).

The selected artifacts were manually analyzed by six of the authors with the goal of classifying them as false positive (*i.e.,* unrelated to documentation issues) or assigning a set of *labels* describing (i) the documentation issue discussed, (ii) the solution proposed/adopted, (iii) the type of the documentation and (iv) the documentation tools discussed. Each of these labels was optional. For example, it is possible that only the issue type and the solution were labeled for an artifact.

For two of the four categories, namely issue type and documentation type, we started from a predefined list of labels. For the issue types, we used the 13 quality attributes defined by Zhi *et al.* [ZGYS+15]. For the type of documentation, we had 11 predefined labels that we selected based on our experience (*e.g.,* code comments). See the replication package [doc19] for the complete list of predefined labels.

The labeling was supported by a Web app that we developed for this task and for conflict resolution. Each author independently labeled artifacts randomly assigned to her by the Web app, selecting a proper label among the predefined ones or defining a new label when needed. To assign a label, the author inspected the whole artifact and, in the case of issues and pull requests, also the related commits. Every time an author had to label an artifact, the Web app also showed the list of labels created by all taggers so far. The labeling was performed at **sentence level**: The Web app allowed the author to select one sentence from the artifact at a time and assign labels to it. This means that multiple sentences could be labeled for each artifact and hence multiple labels could be assigned to it. This allowed us to create a (publicly available) database of labeled sentences related to documentation

artifacts (see the replication package [doc19]).

Each artifact was assigned to two authors by the Web app. In case both of them classified the artifact as a false positive, the artifact was replaced with another one randomly selected from the same source (*e.g.,* a false positive email thread was replaced with another email thread). For each artifact X in which there was a conflict in the assigned labels, a third author (not previously involved in the labeling of X) was assigned to solve it. A conflict in this scenario can happen for many reasons. First, the two authors could label different sentences in the artifact. Second, assuming the same sentences are selected, different "categories" of labels could be assigned to the sentences (*e.g.,* one author labels a sentence as discussing the issue, one as presenting a solution). Third, assuming the same sentences and the same categories of labels are selected, the label values differ (*e.g.,* different solutions indicated for the same sentence). Fourth, one author could classify the artifact as a false positive, while the other could label it. For these reasons, we had a high number of conflicted artifacts (765 out of 829 — 92.27%). We solved some specific cases automatically. In particular, if two authors (i) labeled for the same artifact two different sentences S_i and S_j where S_i is a substring of S_j (or *vice versa*), and (ii) had no conflicts between the label values, we automatically solved the conflict by selecting the longest sentence as the valid one. This reduced the number of conflicted artifacts to 592, which were manually reviewed by a third author who could accept a conflicting sentence (and apply minor modifications if necessary) or discard it.

In this final process, 5 artifacts were discarded as false positives. The final number of sentences labeled for each type of artifact is reported in Table 5.1.

5.2.2 Data Analysis

We answer our RQ by presenting a taxonomy of the types of documentation issues found in our analysis.

Such a taxonomy was defined in an open discussion involving all the authors and aimed at merging similar labels and hierarchically organizing them (see Figure 5.1). We focus our qualitative analysis on specific categories of issue types. For each category, we present interesting examples and common solutions, and discuss implications for researchers and practitioners.

5.3 Results & Discussion

As a result of the labeling process, we obtained 824 artifacts including a total of 1,548 labeled sentences.

Figure 5.1 shows the hierarchical taxonomy of the 162 documentation issue types that we identified. They are grouped into four main categories: (i) problems related to the *information content* of the documentation describe issues arising from "what" is written in the documentation; (ii) issues classified under the *information content (how)* category focus on "how" the content is written and organized; (iii) the *process-related* category groups issues related to the documentation process; and (iv) *tool-related* matters originate from the usage of a documentation tool. The number shown in the main categories of Figure 5.1 represents the number of artifacts related to that issue (*e.g.,* 81 artifacts were related to process-related issues). Note that a single artifact might discuss multiple types of issues. Figure 5.1 also shows the distribution of the analyzed artifacts among the four sources we analyzed. Interestingly, problems related to the content and how it is presented/organized are mostly discussed in issues and pull requests; and discussions about the documentation process and tools-related issues are mainly held in mailing lists and SO respectively.

For each category, we next describe representative examples and discuss implications for researchers (indicated with the ⚗ icon) and/or practitioners (🔧 icon) derived from our findings.

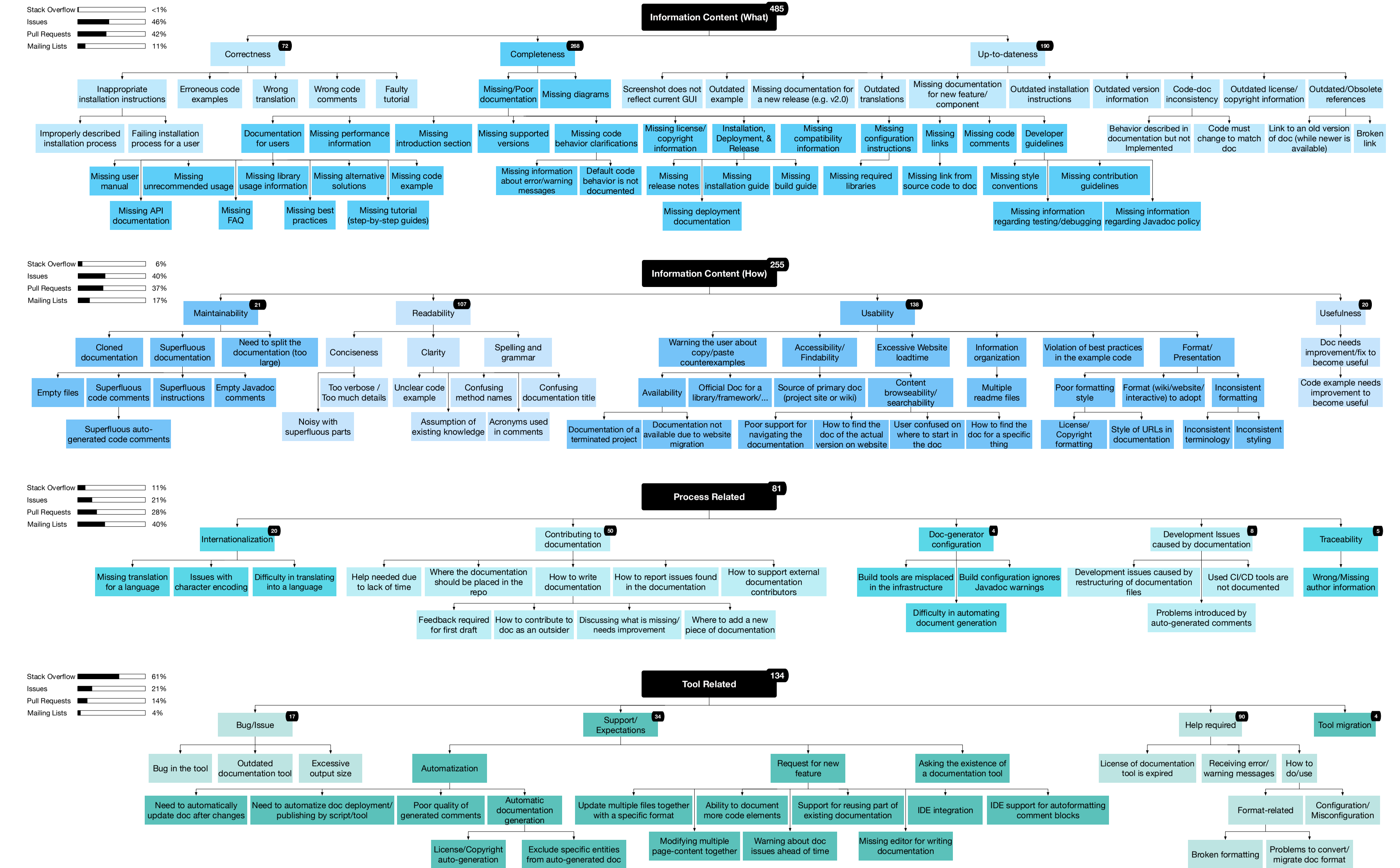


Figure 5.1. Documentation Issues Taxonomy

5.3.1 Information Content (What)

A total of 485 artifacts discuss issues related to the information content, *i.e.,* "what" is written in the documentation.

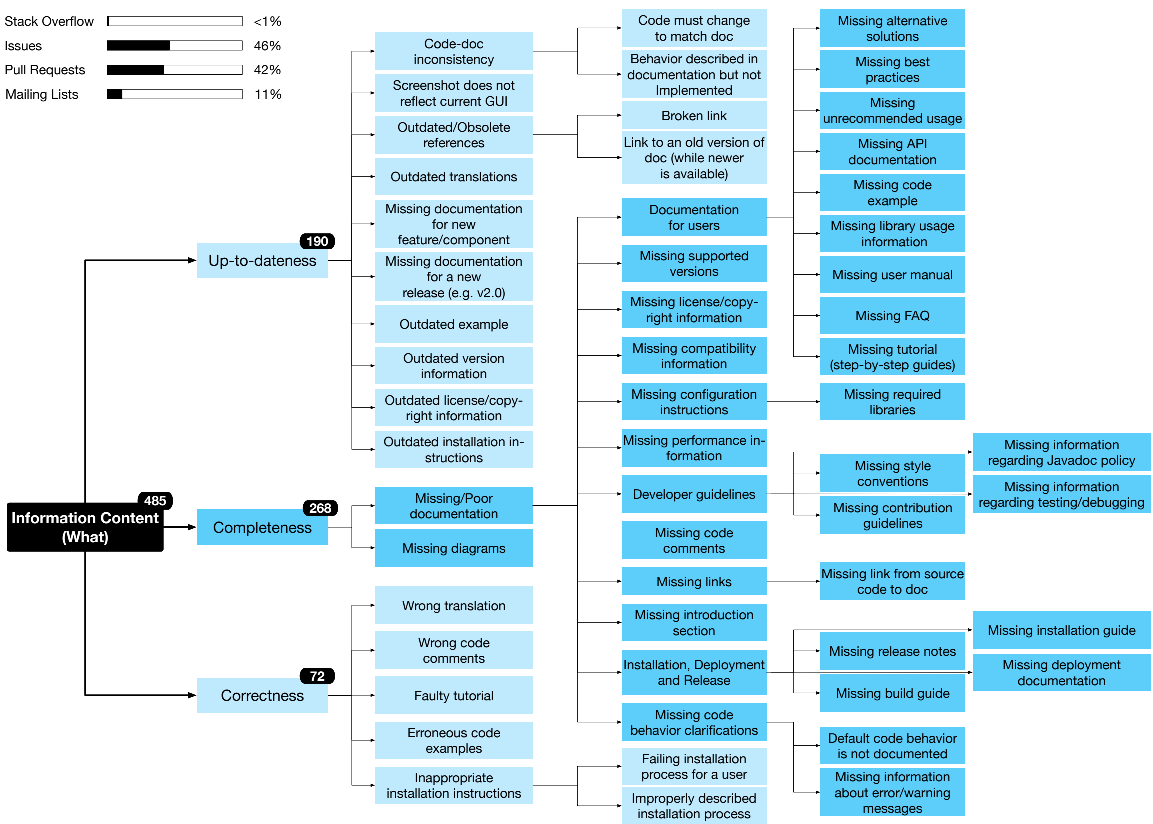


Figure 5.2. Documentation Issues related to information content (what)

Correctness (72). Correct documentation provides accurate information in accordance with facts [ZGYS$^+$15]. Incorrect documentation might have unforeseen serious consequences, going beyond wasted time trying to replicate a wrong code example or following the wrong steps in a tutorial. This is the case of an issue filed for the *acid-state* project, a tool providing ACID guarantees to serializable Haskell data structures. As reported in the issue, a false claim in the documentation could lead to data loss: *"This could easily cause permanent data loss if the user then proceeds to remove the Archive folder, which is claimed to be safe by the documentation"* [1].

The type of documentation most frequently impacted by correctness issues was code examples (*e.g.,* [2]), accounting for 50% of the cases in which we labeled a documentation type, followed by installation guidelines (20%, *e.g.,* [2]). Correctness issues in code examples include syntactic mistakes (*e.g., "the documentation gives the following example [...] but running it leads to ERROR: syntax: [...] is not a symbol"* [3]), as well as more serious programming errors (*e.g., "one of the example fixture files in the documentation would not work because it contains references to objects that have not yet been declared"* [4]). In general, due to their potential consequences, correctness issues were handled with care by developers. For example, we found a case of correctness issue caused by a wrong translation, where developers not only fixed the mistranslation but also decided to have the document reviewed by a native speaker [5].

Completeness (268). Documentation is incomplete if it does not contain the information about

the system or its modules needed by practitioners/users to perform their tasks [ZGYS+15].

Completeness accounts for 55% of the issues related to the documentation content. We observed different causes of incompleteness. For example, in an email sent to the *Apache httpd* mailing list, a user complained about missing definitions of ambiguous terms: *"is there any idea what "frequently" might mean?"* [6]. Indeed, the documentation states *"[...] should result in substantial performance improvement for frequently-requests files"*, without providing a clear definition of what "frequently-requests files" are. Other common completeness issues are related to missing descriptions of library components (*e.g., "[...] missing information about the toolbar buttons"* [7]), missing API usage clarifications (*e.g., "I think that we should add documentation ensuring that the user passes a tree with reset bounds"* [8]) and lack of compatibility information (*e.g., "Explicitly mention if clang 4.x, 5.x are supported"* [9]).

It is worth mentioning that API references and code comments are the types of documentation mostly affected by completeness issues.

Up-to-dateness (190). A document is outdated when it is not in sync with other parts of a system. Up-to-dateness differs from "Correctness" and "Completeness" in that the information was correct and complete before a change was introduced.

Up-to-dateness problems account for 39% of issues related to documentation content. In 21% of these cases, the inconsistency appeared to be between a system's behavior and its description in the documentation. The discrepancy was usually triggered by a change in the code that required to change parts of the documentation or to add/remove content. This latter case typically happened when new features are implemented, *e.g., "include documentation around the new field converter feature"* [10]. In other cases, instead, users complained about the documentation of a behavior or functionality that became unavailable (*e.g., "the setLeftScale and setRightScale routines mentioned in the doxygen documentation seem not to exist"* [11]).

While most of the times it is the documentation that does not reflect what is implemented in the code, in other cases it is the code that needs to be updated to match the documentation. For instance, implementing a method in a non-thread-safe manner was questioned by a user as *"the callback is not thread-safe which it has to be according to the documentation"* [12]. In another case, the required change was minor (*e.g., "slight change to strings in the admin console to reflect the documentation"* [13]) but still needed to ensure consistency between code and documentation.

There were also situations in which there was a debate to decide whether the code or the documentation needed adjustment to fix the inconsistency. This was the case of a GitHub issue related to an inconsistency between the documented and actual behavior of an API: *"Is this an error in the code, or an error in the documentation?"* [14].

Referring to deprecated information is another reason for up-to-dateness issues and can affect the documentation in different ways. It includes having deprecated information in the project's website (*e.g., "homepage recommends deprecated commands"* [15]), outdated copyright information [16] and version numbers [17] in the code base, as well as outdated references (*e.g.,* links to old versions of the system), which was the most prevalent issue within this category. For example, a user reported that *"the example linked in the documentation is using the 3.x version of the API, and that may be confusing to readers"* [18]. In another example, an outdated link in documentation was removed by developers since the target no longer existed [19].

There were also some cases in which it was necessary to rewrite the whole documentation for a major new version [20].

Some developers adopted preventive solutions to ensure documentation up-to-dateness, adding this as one of the items to check in the contribution to-do list [21], or even making Javadoc update mandatory for pull request acceptance [22].

Discussion and Implications

Our results highlight frequent issues related to the correctness, up-to-dateness and completeness of the information reported in the documentation. The documentation types most frequently affected by correctness issues are, not surprisingly, code examples. Indeed, as it happens with production code, bugs can affect code examples as well. A recommendation to mitigate this problem is to apply testing techniques on code examples as done on production code. However, this might not be trivial since documentation often reports incomplete examples rather than entirely runnable programs (*e.g.,* a snippet of code on how to use an API is shown, but the snippet cannot be actually compiled and run).

Devising approaches to (i) test complete/incomplete code examples in documentation and (ii) validate the consistency between snippets and source code is a research challenge for the software engineering community. Assuming the availability of such techniques, regression testing of code snippets could be performed to ensure they are always up-to-date. A more challenging scenario is to automatically generate code examples to be included in documentation.

Up-to-dateness and completeness issues can also benefit from careful traceability of information between documentation and code. We observed issues related to documented code that does not exist in the system anymore. To address this issue, both developers and researchers should take action.

Developers, on the one side, should keep track of documented/undocumented code components. One way of doing so is to use a contingency matrix, where rows represent code components and columns represent existing documentation artifacts. A check in the $\text{entry}_{i,j}$ would indicate that the component i is documented in the artifact j. This matrix can then be queried to check for inconsistencies.

Researchers, on the other side, should continue their work on traceability link recovery [ACC+02], investing in the implementation of tools that can be easily adopted by developers.

Finally, some of the issues we observed (*e.g.,* ambiguous terms in the documentation) highlighted the importance of including documentation users in the loop. Indeed, information that might look clear from the developers' perspective is not always easy to digest by the users of the system. Involving them in the review of the documentation might help in minimizing the users' learning curve and in avoiding misunderstandings. For example, in the case of libraries, developers of the client projects might be invited during code reviews involving substantial changes to the documentation (*e.g.,* when a new release is issued).

5.3.2 Information Content (How)

A total of 255 artifacts discuss issues related to *how* the content of the documentation is written and organized.

Usability (138). Usability of documentation refers to the degree to which it can be used by readers to achieve their objectives effectively. This category covers issues affecting users' experience with the documentation.

Half of the issues (50%) were related to information findability, *i.e.,* when the desired information was available but couldn't be found by a user, *e.g., "I cannot find the description or implementation notes"* [23] or *"I can't seem to find the API documentation anywhere. Could you please host it somewhere or point me there"* [24]. Developers often handle these issues by providing users with pointers

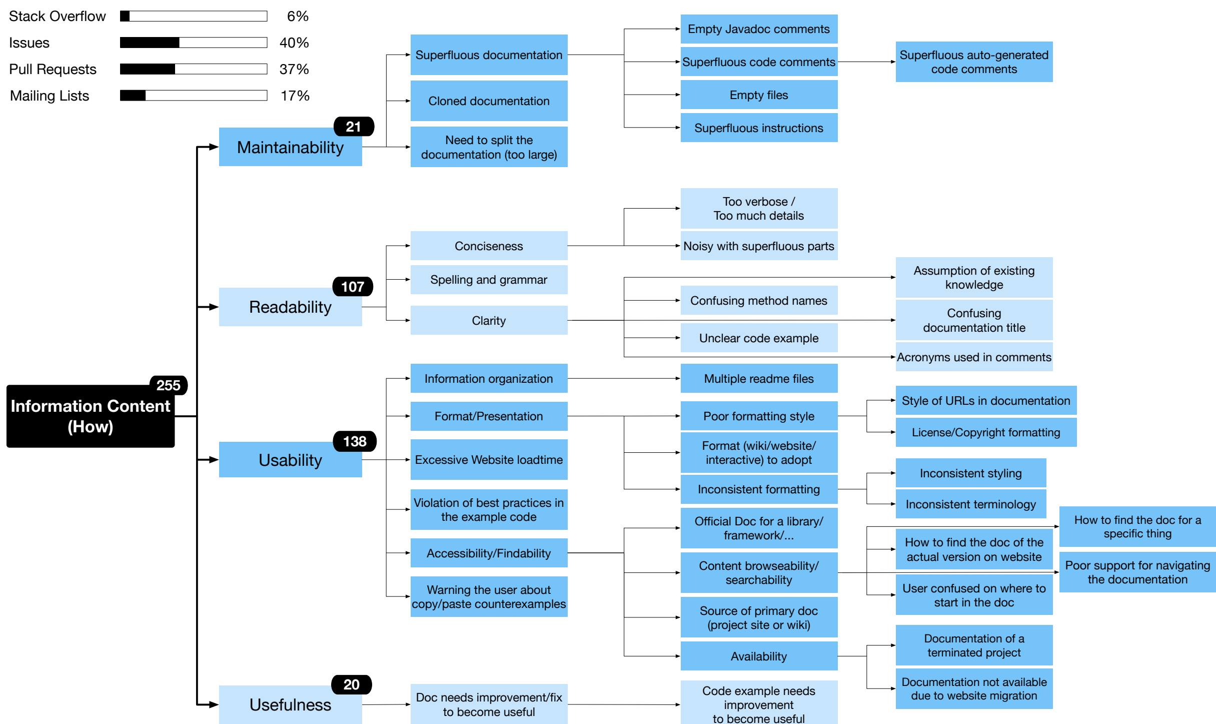


Figure 5.3. Documentation Issues related to information content (how)

to the documentation needed. In some cases, they go further to improve the user experience by implementing a search feature in the project's website [25] or by adding more intra-documentation links [26].

Information organization (18%), *i.e.,* how intuitively and clearly the information is organized [ZGYS+15], constitutes the second most common concern in this category. Placing documentation in standard locations is an effective practice to help users locating it, *e.g., "the consolidated document [...] is compiled into the 'docs/' folder, because as you already said, this location is much more prominent and easier to find"* [27]. Moreover, leveraging intra-documentation links for easier navigation [27], preparing a template (*e.g., "I have setup a page template that can be used as starting point for new pages"* [28]) and adding a 'Table of Contents' for easier navigation [29] were among other popular solutions to ensure a good information organization in documentation.

Poor or inconsistent formatting was another common issue, though not really a barrier for using documentation (*e.g., "heading styles should be improved to have a better separation between H1, H2"*). In general, users only complain about formatting when other types of usability issues emerge (*e.g., "I never find what I want on revapi.org [...] the link structure is counter-intuitive, some links are somehow hidden"* [30]). We also observed intra-documentation consistency issues (*e.g., "inconsistent title between sidebar and article"* [31]) and problems related to poor content organization (*e.g., "the order of the modules on modules.html is pretty arbitrary"* [32]).

Availability, *i.e.,* whether the documentation is accessible, was also an issue in the analyzed artifacts. For example, in response to a user who was looking for the documentation of a plugin, a developer answered *"unfortunately, at the moment not all documentations for all plugins have been migrated yet. This is currently under going"* [33]. In another case where a user was looking for documentation of a terminated project[33], web archive services (*e.g.,* Wayback Machine[1]) were

[1] See `https://archive.org/web/`

suggested.

Maintainability (21). This category concerns issues related to the maintenance of documentation, *e.g.,* how easy it is to apply changes or corrections to it. Just like in source code, duplicated content caused troubles for documentation maintainers, which were mostly resolved by replacing the clone with links and references. However, we noticed that documentation frameworks often make it hard to avoid duplicates. For instance, due to the document format requirements of *Jazzy*[2], a documentation tool for Swift and Objective-C, users have to create unwanted duplicates for *Xcode Quick Help* (an IDE feature for showing methods' comments), as a developer reports: *"duplicating the documentation is admittedly annoying, but that's still the only thing that satisfies Quick Help"* [34]. In another scenario, due to format mismatch between GitHub pages and *AsciidoctorJ*[3], a Java documentation tool, the documentation content was kept in two locations: *"the reason for these 2 locations is that GitHub does not resolve the includes"* [27].

Another noticeable issue was the existence of superfluous files that might cause confusion. This was suggested by a developer of the *Apache httpd* project: *"get rid of these no-content files so they don't confuse the issue of what still needs to be documented"* [35].

The way information is organized and modularized was another concern affecting maintainability as mentioned by a developer *"The documentation has also been pared back a bit, mainly to make it easier to maintain between changes"* [36].

Readability (107). Readability is the extent to which documentation is easy to read. Issues related to lack of clarity represented more than half (55%) of these problems. A user of the *Apache stdcxx* project complained: *"we were able to solve the problem using the information in the users guide, but [...] the documentation is rather confusing on exactly how this needs to be set"* [37]. Abstract [38], too technical [39] and too verbose/noisy [39] information were among the main reasons for poor readability. Developers reacted to these issues by rewriting unclear parts of the documentation, *e.g., "this pull request aims to better explain the differences between these two options"* [40] or *"I don't know if this is the best wording, but I found this behavior confusing and not clearly explained in the docs. Hope this clarifies things a bit"* [41].

The second most frequent cause for readability issues were simple typos. Fixing such errors was always welcome: *"language corrections would be a hugely appreciated contribution too"* [38], especially when they affected user documentation: *"[...] we do not have to be as stringent as we have to be for user visible docs"* [42].

Rewriting a part of the documentation from scratch [39] and improving the description of code examples [43] was also adopted in some cases to fix readability issues.

Usefulness (20). A document is useful if it is of practical use to its readers, *i.e.,* readers can successfully achieve their goals with the help of the document. Depending on the reader's goal, usefulness can be affected by several factors. For instance, in response to a documentation update, the owner of a project suggested: *"it would be good to make this example a bit more realistic"*. In this case, the code example is neither outdated or wrong, but it required improvements to be more useful.

Many maintainers addressed usefulness by asking users' feedback on the documentation. In one scenario, developers collected user feedback to improve the documentation website in two steps: first they conducted a survey prior to documentation refactoring, and then they gathered feedback to ensure that the changes met the users' needs: *"we did a survey prior to the doc lockdown to get an idea of what we should focus on. Now we have a yes/no style survey to ensure that we met the user needs when it came to improving the docs"* [44].

[2] See `https://github.com/realm/jazzy`

[3] See `https://github.com/asciidoctor/asciidoctorj`

Discussion and Implications

Besides the information content (*i.e.,* what is in the documentation), the way it can be consumed (*i.e.,* how effectively its content can be exploited) strongly influences documentation quality. As our analysis reveals, a major part of the discussed issues is related to the usability of the documentation, stemming from poor information organization and findability issues (if existing documentation cannot be found it conceptually does not exist).

Developers can prevent/address these issues by: (i) providing a search engine in the project's website to improve content findability; (ii) adopting a consistent documentation format, *e.g.,* a template that ensures the presence of intra-documentation links and a table of contents, or a style similar to existing documentation recognized by its quality (*e.g.,* see the MongoDB documentation[4]); and (iii) archiving the documentation of old, dismissed versions of their projects in a specific location, to make them available to users who cannot update to newer versions.

We also support the idea adopted by some developers to survey their users [44] to investigate their documentation needs.

In this context, researchers can contribute to improving documentation quality by working in two directions. First, in the same way that code clone detection techniques have been implemented [Kos07], approaches to detect documentation clones and automatically remove (refactor) them could help developers in reducing redundancy in documentation. Second, similar to code readability metrics [SLOP18], readability metrics tailored for documentation could help developers in spotting and fixing readability issues. While one may think that software documentation consists only of text and, as such, standard readability metrics for text can be used (*e.g.,* the Flesch-Kincaid readability formulas [KFRC75]), software documentation is often a mix of text and code that uses domain-specific terms. Moreover, while part of the problem is to classify a document as highly/poorly readable, a more difficult challenge is to indicate to the developer the exact section of the documentation causing the readability issue. Thus, creating "documentation linters" is an interesting avenue for future research.

In addition, studies investigating the users' behavior when looking for documentation could help to define better practices for the organization and presentation of documentation.

5.3.3 Tool Related

In this section we discuss four types of issues related to documentation tools (*e.g.,* Javadoc) found in 134 artifacts.

Bug/Issue (17). This category refers to problems presented by documentation tools (*e.g.,* bugs, malfunctions) that are not originated from improper usage or configuration.

Bugs in software systems are quite common, and documentation tools are no exception. For instance, we found a case where a bug in *Doxygen* installer caused troubles for a user (*i.e., "a glitch in the Doxygen installation script which caused installation failure"* [45]), and the bug was fixed within one day of being reported to the project's mailing list.

Users often asked questions on SO when they were not sure whether they experienced a problem originating from wrong usage or a bug in the tool. These stories usually ended up in the issue tracker. In one case, when a user failed to parse a markdown file with *Doxygen* asked a question on SO saying

[4] See `https://docs.mongodb.com`

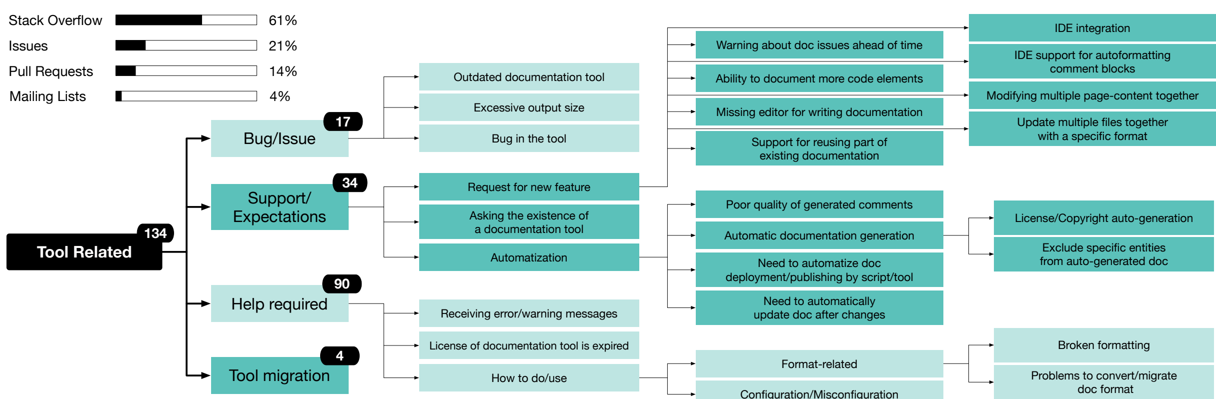


Figure 5.4. Documentation Issues related to tools

"By this definition, this should work [...] Is this a bug, or am I not doing it right?". She got a prompt response: *"This appears to be a bug in the Markdown parser you are using. You might consider reporting it to the developers of that project"*.

In another case [46], a user noticed that *stack haddock*[5], a toolset for Haskell development, does not generate documentation for the dependencies of the executable or test components, but only for library components. Although the issue is still unsolved and might look like a feature request, it is labeled as "Should" by developers, which implies that the current behavior needs to be changed.

Support/Expectations (34). This category covers developers' needs that were not fulfilled by documentation tools, such as missing features (*e.g., IDE integration*).

Users often wanted popular tools to be available in several contexts/languages. Examples of such requests are: *"Is there anything like GhostDoc for C++"* [47], or *"Is there any tool which provides these Doxygen-style features for Ruby?"* [48].

New features for existing tools were requested several times, as in a scenario where a user needed quick access to Android documentation from Android Studio [49].

Automatization was also frequently discussed. A representative example is the automatic deployment of documentation, which was implemented in a project after a developer's suggestion to automatically publish the latest documentation for a specific branch [50].

Help required (90). This category covers issues caused by improper tool usage or configuration rather than by bugs.

Warning and error messages from documentation tools were discussed in all analyzed sources. Examples of these warnings/errors were related to the use of a wrong Python version [51], a wrong path in the documentation build configuration [52], inconsistencies in the Javadoc comments format [53] and empty Javadoc code comments [54]. Providing tool usage examples was the most prevalent solution (*e.g.,* [55]).

"*How to*" questions related to documentation tools, *e.g., "how can I get Sphinx to recognize type annotations?"* [51], account for 83% of the observed issues in this category. This type of question was mostly asked in SO (79%). Formatting was a major sub-issue. In one case, *phpDocumentor*[6] generated files with a wrong format: *"When running phpDocumentor, the resulting files/folder looks extremely weird"*. This issue has been open since November 2015 [56].

Tool migration (4). This category refers to issues related to migration, either to a newer tool version or to another tool.

[5] See `https://github.com/commercialhaskell/stack`
[6] See `https://www.phpdoc.org`

Errors after migrating to a newer version of the same tool were observed in two out of four discussions we analyzed. In one scenario, a user faced numerous errors with a newer version of Javadoc: *"javadoc is having troubles compiling Tools and I can't see why. It has only happened since I migrated to Java 8. I never saw this issue with Java 7"* [57].

In another case, developers noticed that a navigation bar disappeared from the documentation after migrating to a newer *Sphinx*[7] version. The answer noted that this was a change in the default theme, but could be set to behave as it did before.

Discussion and Implications

Most of the tool-related issues we identified can be generalized to issues experienced by users with any type of tool, not just with the one related to documentation. The prevalence of "*how to*" questions in this category, reinforce our findings on completeness and findability of documentation (see Section 5.3.1). Indeed, these questions are likely the result of missing (or difficult to find) documentation in documentation tools. Thus, the same previously distilled implications for researchers and practitioners apply here.

We identified many artifacts discussing feature requests or tool expectations, which carry a message for practitioners to pay attention to common needs, such as support for IDE integration, handling of multiple documents together and automatic document/comment generation.

In addition, researchers could develop approaches to help users in understanding whether a problem they are experiencing is due to a tool misusage or, instead, if it is a well-known bug of the tool. This can be done, for example, by capturing characteristics of the error (*e.g.,* the generated stack traces if available) to automatically search on the project's issue tracker and/or on Stack Overflow for related discussions.

5.3.4 Process Related

Documentation process issues are discussed in 81 artifacts.

Internationalization (20). This category covers issues related to translation processes, *e.g.,* missing/wrong language translations, the need for reviewing translated documents and rendering problems due to character encoding.

The lack of translated documentation was a recurrent problem (*e.g., "is there a danish translation started?"* [58]), especially when the English documentation was not available, which represented a usage obstacle for several users. This was the case of a project mainly documented in Chinese. In a pull request created to start an English version of the documentation, the main developer apologized for the lack of translation: *"Most user are Chinese, include me. Our English is not good, so sorry"*, to which a user replied *"I could use google translate, but the more effort I have to put into understanding a framework, the less likely it is that I use it"* [59].

Many projects benefited from crowdsourcing the translations, which allowed external users to contribute. To this end, projects that did not have the documentation on code-sharing platforms, discussed whether to move the documentation to obtain more contributions: *"Wonder [...] if we shouldn't push our doc on GitHub to ease contributions"* [60].

Missing guidelines on how to contribute a translation was also a common concern, mostly addressed by providing a page with instructions, *e.g., "we need a webpage describing the basics of how to go about translating the apache docs"* [39].

[7] See `www.sphinx-doc.org`

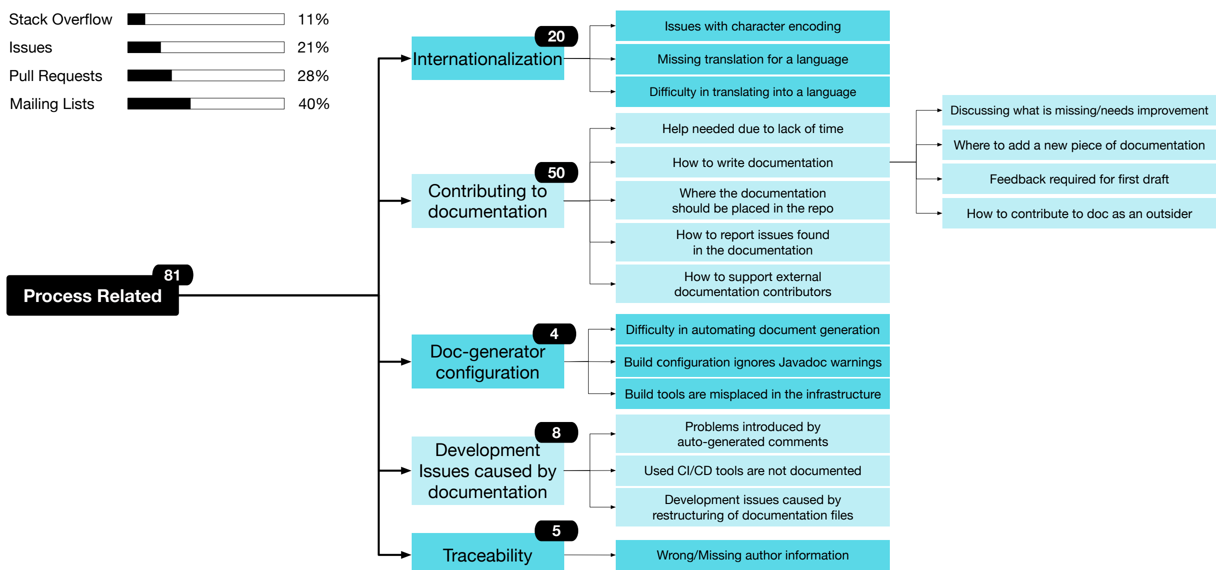


Figure 5.5. Documentation Issues related to processes

Character encoding was another typical source of problems in the context of document translation, as developers were often puzzled about the proper encoding to choose. For example, in a mailing list discussion for the simple question *"Which encoding should be used for the .fr files?"* [61], several encodings were suggested, because factors such as file size or encodings better supported by clients were considered.

Traceability (5). This category concerns issues related to the ability to track documentation changes, *i.e.,* to determine where, when, by whom and why a change was performed.

A straightforward solution to keep track of changes in a document was to manage it with a version control system, *e.g., "move wiki to /docs [...] It also means docs are versioned with each new release"* [62]. It is not always feasible, however, to track changes in a version control system. Examples include cases in which the documentation is stored in binary format or in a database. In such cases, preserving traceability by at least versioning some meta-information for each document was a common solution: *"We need a webpage describing [. . .] and perhaps some standard comments to put at the top of each doc (english version, author, reviewer)"* [39].

Development issues caused by documentation (8). This category covers issues caused by documentation, *i.e.,* unwanted effects of documentation on the development process.

In one interesting case, auto-generated documentation caused issues for the reviewing process of pull requests, as it resulted in noisy diff outputs. As a solution, the developers suggested to split the documentation and the code changes into two separate commits: *"It would be great if we could find a way [...] to defeat the generated files from showing up in the PR diffs, as they overwhelm the diff and make it very hard to review for any other changes. To that end, it would be great if this PR could be squashed into two commits (one with script etc changes, and one with only generated docs"* [50].

Contributing to doc (50). This category covers issues encountered by (internal or external) contributors of documentation while they were reporting/fixing errors or writing new documents. It also includes developers' concerns on supporting external contributors.

We observed that many projects welcome contributions to their documentation from non-members of the projects. To do so, they tried to facilitate the contribution process by offering different aids, as an Apache developer said *"I've tried to lower the barrier [...] to allow anyone to contribute. You can now edit and review change[s] to the jclouds.incubator.apache.org site entirely within your web*

browser.” [63]. In one scenario, a developer opposed involving a less-known technical solution, namely GitHub pre-commit[8], into the contribution pipeline and said *“I am reluctant to use precommit hooks to modify the document, as it makes contributions from the community more difficult”* [27]. Indeed, a non-documented and more laborious contribution process can make someone back out of contributing. For instance, a user who wondered how to update an incomplete documentation page stated: *“I don’t know where I should modify this page, I have no problem to update it but because I know that cannot be modified directly I don’t know where to do it”* [64]. To avoid this situation, well-explained contribution guidelines [65] were provided, sometimes even augmented with a documentation template [28].

As another example, organizing documentation files was also proposed in a discussion thread as a good practice for easing up contribution process: *“We also might want to think about splitting this into chapters so that it’s easier to work with.”* [27].

Another common issue was related to the lack of knowledge about best practices to write code comments or documentation, *e.g., “how can I document this in JSDoc return type”* [66]. In another example, a user who started a documentation page, sent an email to get feedback on the draft version of the document by adding *“I have just started some user-guide type of documentation [...] Any feedback is welcome”* [67].

Moreover, we found organizing/structuring documentation content another issue. For example, a user who created documentation for a module in a project wrote *“Not sure about location of this one into the doc, but put into administrator.”* [68]. Likewise, from a higher level of abstraction, deciding on where to place documentation files on a repository was the issue in some cases, *e.g., “[...] how the old and new documentation can coexist within the same repo [...]”*.

Identifying missing information (*e.g., “we’ll need to get more info on what is missing”* [69]) was also observed in some cases. In order to report documentation errors, creating a patch was among mentioned methods, *e.g., “Report them together with a patch that fixes them.”* [70].

Doc-generator configuration (4). This category covers issues related to documentation generators, mostly found in the context of the building process of a project.

An important issue was observed in a thread of the *Apache SystemML* mailing list, where a developer complained about incomplete and outdated API comments due to the project’s build configuration that ignores the warnings of the documentation tool[71]. To improve the documentation quality, the developer suggested marking these issues as blockers, with the goal of fixing them in the next release. In another project, developers decided to treat documentation issues warnings as errors, making the build fail: *“So now once warnings are fixed, maybe we could change them into errors, so when somebody makes a mistake it will cause build to fail”* [72].

Discussion and Implications

Many of the issues related to the documentation process concern the way in which external contributors can help in writing, updating and translating documentation. Our findings can be distilled into guidelines for developers to ease the documentation process, which can result in higher-quality documentation and pleased contributors.

Developers, first, have to provide contributors with clear guidelines (ideally accompanied by documentation templates) that carefully explain what is expected to be covered in the documentation, how different types of documentation (*e.g.,* code comments) should be written and what the process to contribute is.

[8] See `https://pre-commit.com/`

Second, developers have to consider widely-used code-sharing platforms (*e.g.,* GitHub) to host documentation, where the likelihood of attracting external contributors from all around the world is quite high, which could help with time-consuming tasks such as the translation of documentation.
Third, developers have to adopt mechanisms to promote good documentation practices, such as making a build fail when documentation issues are spotted via program analysis (*e.g.,* a new method has been implemented but one of its parameters has not been documented in the Javadoc).

Another take-away for developers is the need to provide English documentation for their software projects, which would increase their adoption (and, possibly, the contributions).

Researchers have instead the possibility to work on the optimization of these documentation processes and answer fundamental research questions, such as what constitutes a good contributors guideline (*e.g.,* by surveying software developers).

Finally, as already observed in the literature [RMT+17], advances in the automatic software documentation field are clearly needed. For example, while current static analysis tools used in continuous integration perform simple checks on documentation (*e.g.,* to identify missing comments), the development of approaches that are able to detect more complex *documentation smells* [ZS13] at building time is worthy of investigation.

Open sourcing a software package or some parts of it, such as the documentation, can open possibilities in front of a broader community, which is one of the main reasons why practitioners decide to go open.

This opens the possibility for researchers too to study these processes. Data, such as the sources we used in our study, mailing lists, bug reports, pull requests, forum discussions, has become available for large projects over long periods of evolution courses, which was not the case a few years ago. Researchers should take this opportunity to study this data and help practitioners in optimizing their documentation processes.

The usage of automatic documentation generator tools has become an integral part of the documentation process, and we observed many issues related to their use in the process.

Although these tools have been around for a while, there are many open challenges ahead of them. As currently they are mostly used to generate API descriptions, it is a great challenge for the researchers research community, *e.g.,* to apply recent advances in automatic translation to support internationalization processes, or to take advantage of information retrieval techniques to help supporting consistent documentations or supporting dynamic document generation, such as the idea of on-demand developer documentation [RMT+17].

5.4 Threats to Validity

Threats to *construct validity* relate to possible measurement imprecision when extracting data used in our study. The automatic mining of developers' documentation discussions based on keywords-matching mechanisms resulted in the retrieval of some false positives (as reported in Table 5.1). These imprecisions were discarded during our manual analysis, thus they did not affect our findings.

In our manual analysis, we based the classification of discussions on what was stated in the analyzed artifacts. It is possible that the information reported in individual artifacts is incomplete,

for example due to the fact that an issue was partially discussed in the mailing list and partially via chat.

Threats to *internal validity* concern confounding factors, internal to our study, that can affect the results. They are related to possible subjectiveness introduced during the manual analysis. We mitigated this threat by making sure that each discussion was independently analyzed by two authors and that conflicts were solved by a third author.

Threats to *external validity* represent the ability to generalize the observations in our study. While we analyzed data of different software projects and from diverse data sources, it is possible that our taxonomy of documentation issues depends on the particular set of discussions we analyzed, and that in other contexts developers discuss issues we did not encounter.

5.5 Summary and Conclusion

We inspected 878 artifacts from four different sources to derive a taxonomy of 162 types of issues faced by developers and users of software documentation. We qualitatively discussed our findings and expose implications for developers and researchers, with the goal of highlighting good practices and interesting research avenues in software documentation.

In essence, our study empirically confirms and complements previous research findings (and common sense): Developers (and users) prefer documentation that is correct, complete, up to date, usable, maintainable, readable and useful.

Given the undeniable value of good documentation, the question is why it is and remains unpopular in software development. We believe that the issues unveiled through our study corroborate, on the one hand, the need for the realization of a vision like the one laid out by Robillard *et al.* [RMT+17]: Systems should be capable of documenting themselves automatically. On the other hand, this requires researchers and practitioners to accept the fundamental notion that documentation is not a mere add-on to any software system, but a part of the system itself.

6

Software Documentation: The Practitioners' Perspective

IN THEORY, (good) documentation is an invaluable asset to any software project, as it helps stakeholders to use, understand, maintain, and evolve a system. In practice, however, documentation is generally affected by numerous shortcomings and issues, such as insufficient and inadequate content and obsolete, ambiguous information. To counter this, researchers are investigating the development of advanced recommender systems that automatically suggest high-quality documentation, useful for a given task. A crucial first step is to understand what quality means *for practitioners* and what information is actually needed for specific tasks.

We present two surveys performed with 146 practitioners to investigate (i) the documentation issues they perceive as more relevant together with solutions they apply when these issues arise; and (ii) the types of documentation considered as important in different tasks. Our findings can help researchers in designing the next generation of documentation recommender systems.

Structure of the Chapter

- **Section 6.1** provides the motivation for this chapter.
- **Section 6.2** presents the study design.
- **Section 6.3** and **Section 6.4** discuss our findings.
- **Section 6.5** presents the threats that could affect the validity of our findings.
- Finally, **Section 6.6** concludes this chapter.

Supplementary Material

All material and data used to run the study in this chapter, as well as the developers' anonymized answers are publicly available in our replication package [doc20]. Specifically, we provide:

- The data used for designing surveys, including:
  - The surveys questions
  - The list of "documentation issues" used in Survey I
  - The definition of "Documentation Types" and "Software-related Tasks" used in Survey II
- The anonymized participants' answers, including:
  - The distribution of participants' countries,
  - Surveys' collected responses (anonymized),
  - Categorization of collected answers based on the type of participants (ABB *vs.* external)

Accomplishments in a Nutshell

Software Documentation: The Practitioners' Perspective [ANVM+19]
Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, David C. Shepherd
In *Proceedings of 42nd ACM/IEEE International Conference on Software Engineering (ICSE 2020)*, To be published. IEEE, 2020.

6.1 Motivation

A “software post-development issue” [fCMA12]. An after-thought, so to speak. This is how the ACM Computing Classification Systems (CCS) categorizes software documentation. Although peculiar, this classification aligns well with the general perception that there are more exciting things to do than documenting software, especially if said software has already been developed.

Studies abound about software documentation being affected by insufficient and inadequate content [Rob09, RD11, UR15, ANVM+19], obsolete and ambiguous information [UR15, ANVM+19, WNBL19], and incorrect and unexplained examples [UR15, ANVM+19], to name just a few issues. In contrast to this rather sad *status quo*, not only are there studies that attest that documentation is actually useful [FL02, CH09, DR10, RD11, GGYR+15], but also it simply makes sense to document software—it is just not an activity enjoyed by many.

Recent research initiatives [Bav16, RMT+17] have advocated for the development of automated context-aware recommender systems that *automatically generate high-quality documentation*, contextual to any given *task* at hand. As a result, a variety of different automated approaches for the generation and recommendation of documentation (*e.g.,* [MAS+13, PBDP+14, MM16, RJAM17, HLX+18]) have emerged. While the creation of such novel systems entails conceptual and technical challenges related to the collection, inference, interpretation, selection, and presentation of useful information, it also requires solid empirical foundations on *what* information is (or is not) useful *when* to developers.

To provide such foundations, we recently performed a study to distill a large taxonomy of software documentation issues [ANVM+19], and inferred a series of proposals for researchers and practitioners. While our taxonomy was promising, it had not been validated by practitioners, making it mostly an academic construction without the much needed reality check. In this chapter, our goal is to juxtapose our taxonomy with the documentation needs and priorities of practitioners. The first contribution is thus an empirical validation of the taxonomy to answer our first research question (RQ):

RQ_1: *What documentation issues are relevant to practitioners?*

Previous studies on documentation that were run using surveys with developers focused either on specific issues, *e.g.,* using and learning APIs [Rob09, RD11, UR15], or were geared towards *generic* activities, *e.g.,* program understanding, development and maintenance [dSAdO05, GGYR+15]. In contrast, our study provides a comprehensive view of the documentation issues encountered by practitioners.

Moreover, since our goal is to further research in the context of documentation recommender systems, the second contribution of this chapter is an insight into the types of documentation that practitioners perceive as useful when confronted with specific software engineering tasks. Therefore, we formulate our second RQ as:

RQ_2: *What types of documentation are perceived as useful by practitioners in the context of specific software engineering tasks?*

To answer these two research questions, we performed two surveys with 146 professional software practitioners. In the first survey, we focused on the documentation issues that practitioners perceive as more relevant, together with the solutions they apply when these issues arise. In the second survey, we studied the types of documentation that practitioners consider important given specific tasks. Most participants (125) are from *ABB*, a multinational corporation active in automation technology, others (21) have been recruited in specialized online forums. The result is a diversified population of practitioners acting in various roles (*e.g.,* developers, testers).

The body of knowledge provided by the findings of these surveys will allow the research community to prioritize the practitioners' needs and to orient future efforts for the design and development of better automated documentation recommendation systems.

6.2 Study Design

The *goal* is to investigate the perception of practitioners of (i) the relevance of documentation issues, and (ii) the usefulness of different types of documentation in the context of specific tasks. The study *context* consists of *objects*, *i.e.,* two surveys designed to investigate the study goals, and *subjects* (referred to as "participants"), *i.e.,* 146 practitioners, 125 employed in *ABB* corporation, and 21 recruited in specialized online forums.

6.2.1 Research Questions

We aim at answering the following research questions:

RQ_1: *What documentation issues are relevant to practitioners?* This research question builds on our taxonomy of documentation issues [ANVM+19], which consists of 162 issues derived from the qualitative analysis of 878 documentation-related artifacts (*e.g.,* Stack Overflow discussions, pull requests). Although our taxonomy seems comprehensive, we did not investigate which documentation issues are actually relevant to practitioners. RQ_1 aims at filling this gap. Knowing the documentation issues that practitioners consider relevant can guide researchers in the development of techniques/tools aimed at identifying and possibly fixing these issues, rather than others not relevant to practitioners. This also inform documentation writers and maintainers about quality attributes of documentation that must be prioritized.

RQ_2: *What types of documentation are perceived as useful by practitioners in the context of specific software engineering tasks?* This research question studies the types of software documentation (*e.g.,* code comment, release notes) that are considered useful by practitioners when performing a specific software-related activity (*e.g.,* code refactoring, debugging). This information is essential in the design of the next generation of software documentation recommender systems, whose goal is to automatically generate documentation customized for a given task [Bav16, RMT+17]. RQ_2 can shed some light on the type of information and documentation needed by practitioners under specific scenarios.

6.2.2 Context Selection: Surveys & Participants

Surveys. Figure 6.1 depicts the flow of our two surveys. The numbered white/gray boxes depict steps in which participants answer questions, and the black boxes represent either static information pages or an activity automatically performed by the survey application to select the next question to ask. The gray dashed box represents a loop of questions/answers performed repeatedly.

Both surveys have been implemented in Qualtrics[1] and start with a welcome page explaining the goal of the study, that the surveys are anonymous, and that they are designed to last *ca.* 15 minutes each. Once the participant agrees to start, both surveys show a form with basic demographic questions. In particular, we ask the participant's role in the software projects they contribute to (*e.g.,* developer, tester) and their years of experience in programming on a four-point scale: <3 years, 3-5 years, 5-10 years, >10 years. Once done with the collection of the demographic information, the two surveys differ (see Figure 6.1).

[1] See `https://www.qualtrics.com`

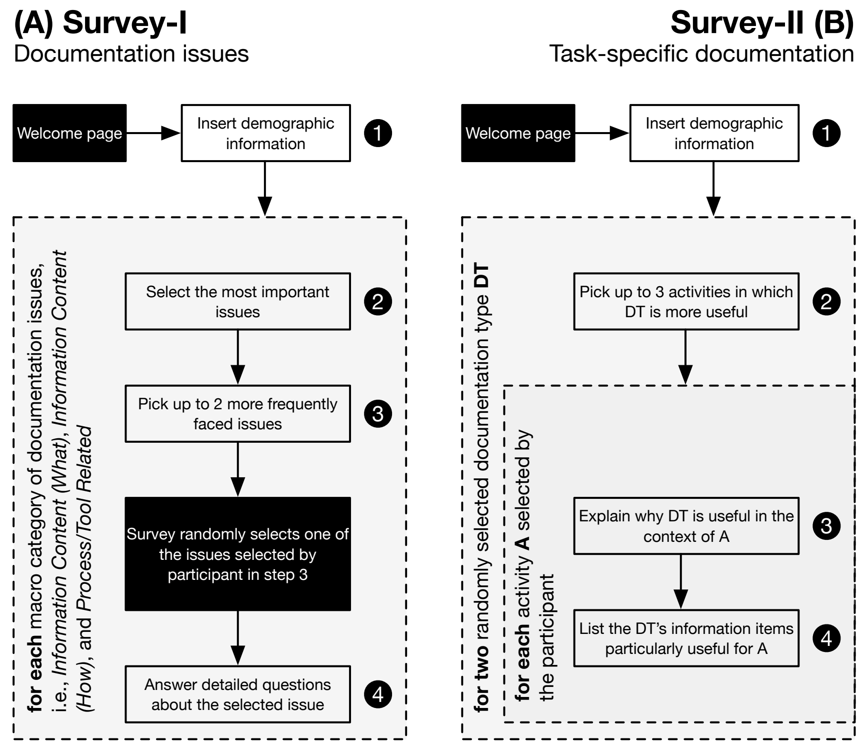


Figure 6.1. Design of the two surveys used in our study

Survey-I. To design *Survey-I* (Figure 6.1-a), we revisited our taxonomy of 162 documentation issues (see Figure 1 in [ANVM+19]) and identified the issues to be considered in our survey. This taxonomy is organized into four categories:

1. **Information Content (What)** refers to problems with the documentation content (*i.e.,* "what" is written in the documentation). This is the predominant category in the taxonomy, with 55% of the analyzed documentation-related artifacts discussing these issues. It is organized into three subcategories: *Correctness* issues (*e.g.,* erroneous code examples), *Completeness* issues (*e.g.,* missing code behavior clarifications), and *Up-to-dateness* issues (*e.g.,* behavior described in the documentation is not implemented).

2. **Information Content (How)** relates to "how" documentation is written and organized, and appears in 29% of the analyzed artifacts. Its subcategories capture issues with different documentation quality attributes, including *Maintainability* (*e.g.,* cloned documentation), *Readability* (*e.g.,* confusing documentation title), *Usability* (*e.g.,* poor support for navigating the documentation), and *Usefulness* issues (*e.g.,* code example needs improvement to become useful).

3. **Process Related** groups issues linked to the documentation process, which were found in 9% of the analyzed artifacts. The issues in this category are organized into five subcategories, namely, *Internationalization* (*e.g.,* missing translation for a language), *Contributing to Documentation* (*e.g.,* unclear how to report issues found in the documentation), *Doc-generator Configuration* (*e.g.,* ignored Javadoc warnings), *Development issues caused by documentation* (*e.g.,* problems introduced by autogenerated comments), and missing *Traceability* information.

4. **Tool Related** refers to issues associated with documentation tools, which were discussed in 15% of the analyzed artifacts. Subcategories include *Bugs* in documentation tools, lack of *Support*, unmet feature *Expectations*, usage difficulties (aka *Help required*), and *Migration* problems across different tools.

The taxonomy is hierarchically organized, meaning that each category (*e.g., Information Content (What)*) contains subcategories (*e.g., Correctness*) further organized into subcategories on many levels (*e.g., Correctness* includes five subcategories, one of which is further organized into two subcategories, leading to a total of four hierarchical levels).

Considering all the types of documentation issues composing the taxonomy [ANVM+19] was not an option for our study. In reality, asking a participant to read the complete list of 162 issues and pick the ones that are more relevant is excessive and would have led to a substantial increase in the survey abandonment rate.

For this reason, we limited the number of documentation issues considered in *Survey-I* by adopting the following process. First, we organized the survey into three parts: *Part I* focuses on issues related to *Information Content (What)*; *Part II* concentrates on *Information Content (How)* issues; and *Part III* investigates *Process-Related* and *Tool-Related* issues, together. Second, given the hierarchical organization of the taxonomy, we grouped together documentation issues that are very similar and share the same parent category, with the goal of reducing the overall number of issues to investigate. For example, in our taxonomy [ANVM+19], issues related to inconsistency between code and documentation (*i.e.,* the code implements a behavior different from the one described in the documentation) are categorized into two subcategories, namely *behavior described in the documentation is not implemented*, and *code must change to match the documentation*. We decided to only consider the parent category, *Code-documentation inconsistency*. This grouping was done by two of the authors, and reviewed in multiple rounds by other three authors until agreement was reached.

The complete list of considered issues is available in our replication package. Overall, we summarized the 162 issues from the original taxonomy into 51 documentation issues: 22 in the *Information Content (What)* category, 12 in the *Information Content (How)* category, and 17 in the combined *Process/Tool Related* category.

For each of the three parts of *Survey-I*, we show the list of related issues to the participant, asking them to select via checkboxes the ones they perceive as major issues (step ② in Figure 6.1-a). When hovering the mouse over the name of a specific issue, its brief definition pops up. We also provide an open field "Others", in which the participant could list documentation issues that were not listed in the predefined options. After that, we show the list of issues selected in the previous step as being important, but this time we ask them to select up to two issues that they face most frequently when reading/writing documentation (step ③). Given this selection, the survey platform randomly picks one of the issues selected in step ③ to collect detailed information about it. In particular, we ask in step ④: (i) whether the specific issue concerns more the readers or the writers of the documentation, with possible choices on a five point scale (*i.e.,* only readers, mostly readers, equally both, mostly writers, only writers); (ii) how frequently the issue is encountered by the participant (possible choices: every day, 2-3 times per week, once a week, less often than once a week, never); (iii) what the solution is for the specific issue (open answer); and (iv) to describe the situation in details (optional, open answer). Note again that steps ② to ④ were performed three times, one for each macro category of documentation issues.

Survey-II. Concerning the second survey (Figure 6.1-b), since our goal is to investigate the types of documentation useful in different software engineering tasks, we had to define the list of documentation types and tasks to consider. In the case of the documentation types, we started again from our taxonomy [ANVM+19]. For each of its 162 documentation issue types, we annotated any type of

documentation mentioned, *e.g.,* from the taxonomy node *Inappropriate installation instructions* we extracted the *installation guide* documentation type. The resulting list was then complemented and refined through face-to-face meetings among three of the authors. In particular, documentation types missing in the taxonomy but known to the authors were added, which led to the final list consisting of: *API Reference*, *Code Comment*, *Contribution Guideline*, *Deployment Guide*, *FAQ*, *How-to/Tutorial*, *Installation Guide*, *Introduction/Getting Started Document*, *Migration Guide*, *Release Note/Change Log*, *User Manual*, *Video Tutorial*, and *Community Knowledge*. For each documentation type, we provided a description and examples of information items usually contained in it. For example, the documentation type *Code Comment* is described as "Code Comments summarize a piece of code and/or explain the programmer's intent"; and the corresponding text for examples of information items is "Comments used to describe the functionality & behavior of a piece of code, the parameters of a function, the purpose and rationale for a piece of code". The descriptions and information items for all the documentation types can be found in our replication package.

Concerning the software engineering tasks, we started from the list of activities reported in the Software Engineering Body of Knowledge (SWEBOK) version 3.0 [BF14]. We went through the SWEBOK knowledge areas (*e.g.,* requirements, construction, maintenance) looking for activities that require, involve or produce documentation. The initial list was discussed by all authors to refine the terms for better comprehension of the participants when reading the survey. The final list of tasks used in *Survey-II* consists of: *Requirements Engineering*, *Software Structure and Architecture Design*, *User Interface Design*, *Database Design*, *Quality Attributes Analysis and Evaluation*, *Programming*, *Debugging*, *Refactoring*, *Program Comprehension*, *Reverse Engineering and Design Recovery*, *Software/-Data Migration*, *Release Management*, *Dealing with Legal Aspects*, *Software Testing*, and *Learning a New Technology/Framework*.

Once we defined the types of documentation and the tasks, we designed the survey. In *Survey-II*, after filling up their demographic information, the participant is shown with the name and description of a randomly selected documentation type DT. The survey asks the participant to select up to three software-related activities in which DT is considered more useful (step ② in Figure 6.1-b). For each selected activity A, two open questions are asked: (i) explain why DT is useful in the context of A (③ in Figure 6.1-b); and (ii) list the information items in DT that are particularly useful for A (④ in Figure 6.1-b). Step ② and the loop including steps ③ and ④ are performed twice for two randomly selected documentation types.

We tested both surveys with four developers and four PhD students to check that the questions were clear and that each survey could be completed within 15 minutes, a duration agreed upon with the partner company. As a consequence of this pilot study, we rephrased a number of questions and set the "thresholds" used in our surveys (*e.g.,* only ask detailed questions about one of the issues selected by participant in step ③ of *Survey-I*).

Participants. We started by collecting answers from the practitioners of the partner company. We invited participants via an email that summarized the goal of the study and contained the link to our surveys. There was a single link to both surveys, but the application automatically assigned a participant to one of them, balancing the number of data points per survey.

As a first test batch, we invited 160 practitioners collecting 21 responses. As no problems were detected, we emailed the invitation to 1,500 practitioners and posted an announcement on the Yammer service of the company. We received 104 additional answers, leading to a total of 125 completed surveys (incomplete surveys were discarded), 65 for *Survey-I* and 60 for *Survey-II*.

Computing a response rate for our study is difficult due to the posted announcement, and to the fact that the survey was conducted over the summer. Assuming that all the 1,660 practitioners received and opened our email, and ignoring that other practitioners were possibly reached through the Yammer service, this would result in a 9.5% response rate, which is in line with the suggested

minimum response rate of 10% for survey studies [GJC+09]. We collected a slightly higher number of responses for *Survey-I* as compared to *Survey-II* (*i.e.,* 65 *vs* 60). This is due to the fact that some participants started the study simultaneously (thus being equally distributed across the two different surveys by the platform), but some of them did not finish the assigned survey, thus unbalancing the final numbers.

We also posted the link to our survey on social websites oriented to developers and programming. This allowed us to collect 21 additional complete answers, leading to the final 146 answers to our surveys, 78 to *Survey-I* and 68 to *Survey-II*. An overview of the surveyed participants and their experience is depicted in Table 6.1.

Table 6.1. Participants roles & programming experience

Role	Population	<3 years	3-5 years	5-10 years	>10 years
Developer	55	12	8	10	25
Architect/Technical Engineer	26	1	1	2	22
Technical Lead	19	0	0	6	13
Test Analyst/Tester/Test Engineer	11	1	0	7	3
Others	35	4	2	4	25
	146	18	11	29	88

6.2.3 Data: Analysis

Analysis. We answer RQ_1 by relying on descriptive statistics. For each of the predefined documentation issues and for those added through the "Other" field, we report the percentage of participants that perceives it as an important concern. We also report on how frequently participants face the issues considered as important, and whether they affect more documentation readers or writers. Moreover, we qualitatively discuss interesting cases shared by participants about real instances of these issues and the solutions they adopted to address them. To analyze the participants' solutions ((4) in Figure 6.1-a), we followed an open-coding inspired approach, where two of the authors independently assigned a tag to each of the 101 answers that described solutions to documentation issues. The tag was meant to summarize the described solution (*e.g., improve project management practices* derived from *"Specifically allocate efforts for documentation in the task planning"*). Conflicts were solved through a face-to-face meeting.

To answer RQ_2, we analyze a heat map (Figure 6.3) depicting, for each activity type (rows), the percentage of participants that indicated each documentation type (columns) as useful in that context. Then, we qualitatively discuss the reasons provided by participants ((3) in Figure 6.1-b) to explain why a documentation type is useful during a specific activity. For each documentation type, we also report the information items listed by participants as particularly useful in each of the software engineering tasks that we investigated.

6.3 What documentation issues are relevant to practitioners?

Figure 6.2 summarizes the responses collected for *Survey-I*. We discuss the practitioners' perspective about the documentation issues listed in the three main categories (*i.e.,* "Information Content (What)", "Information Content (How)", and "Process/Tool Related"). We highlight lessons learned and recommendations for researchers (⚗), and confirm/refute previous findings reported in the literature (🔗).

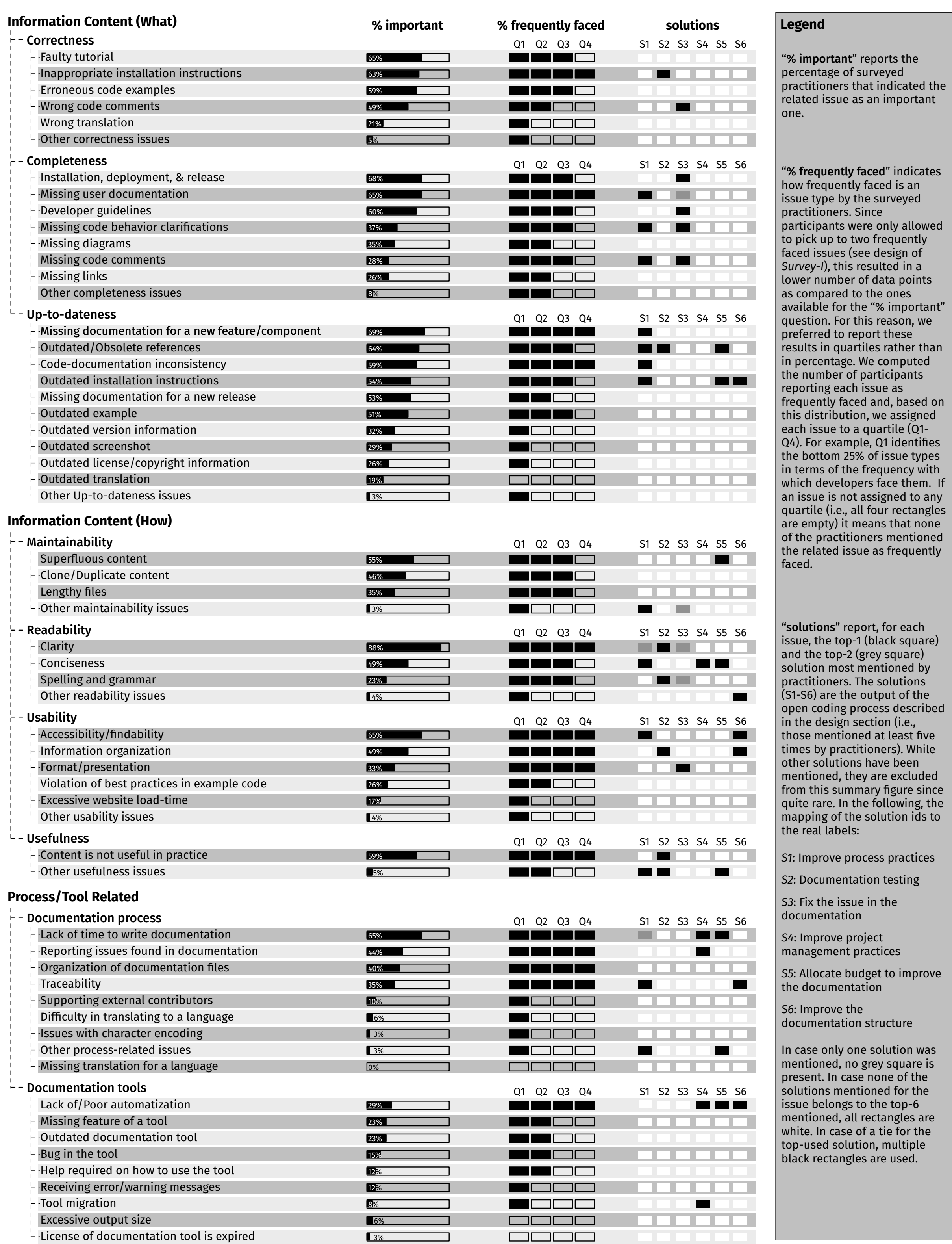


Figure 6.2. Documentation issues relevant to practitioners (RQ_1), according to the results of Survey-I

6.3.1 Information Content (What)

We observe in Figure 6.2 that all issues in this category are perceived as important by practitioners.

This result is in line with previous studies [UR15, CH09] that underlined the relevance of correctness, completeness, and up-to-dateness issues in documentation.

Regarding the *Correctness* subcategory, all its issues except *wrong translation* were considered to be important by at least half of the surveyed participants. Among them, *inappropriate installation instructions* was the most frequently encountered issue and, together with *faulty tutorial*, the one considered relevant by most practitioners (65% of them). Participants suggested a few possible solutions for this issue, such as performing reviews on the installation instructions by both internal team members and external users, who can provide feedback about the quality and usefulness of the document. Lightweight virtualization approaches (*e.g.,* Docker containers) can support practitioners in the creation of diverse reusable deployment environments to test installation instructions.

The fragmentation problem of running environments for software is well-known (see the case of the Android ecosystem [HZF+12, LVMP17]), which might lead to unexpected race conditions or compilation issues under specific platforms [TPB+17a]. Research efforts could be devoted to (automated) testing of installations instructions under different environments, or automated generation of installation instructions. Our survey shows that this is an area of interest to practitioners, and even limited support for testing installation instructions across diverse environments would be welcome.

Wrong code comments is perceived as important by almost half (*i.e.,* 49%) of the surveyed practitioners. Besides the obvious (fixing the comment), practitioners also suggested to train the comments' writers, particularly in their technical English language skills.

Code comments can be incorrect due to inaccurate information, as well as to the writer's inability to clearly describe a code fragment or change rationale in English. This is confirmed by the answers to the *wrong translation* issue. Some participants attributed it to documentation sometimes written by non-native English speakers. A suggested solution is to host the documentation on collaborative platforms (*e.g.,* Wikis) or code-sharing platforms (*e.g.,* GitHub), encouraging and enabling external contributors to add new content or (recommend how to) fix errors in the documentation.

Erroneous code examples were also recognized as an important issue by the surveyed participants (59% of them). It is well-known that code examples are a main information source for developers [RD11]. Facilitating error reporting, *e.g.,* by adding a comment section below each documentation page, was a suggested solution.

We [ANVM+19] suggested the development of testing techniques tailored to code examples. Testing activities on code examples, however, were not mentioned by the surveyed developers.

Among the issues related to *Completeness*, the lack of *installation, deployment, & release* instructions, *user documentation* (*e.g.,* user manual), and *developer guidelines* were considered important by a majority (respectively 68%, 65% and 60%), and are frequently encountered issues. Practitioners highlighted the importance of considering the creation of these types of documentation as first-class citizens, and suggested to include these documentation types as mandatory items in the release checklist and allocate project budget and a dedicated team to fundamental documentation items.

Increasing the budget dedicated to documentation was a recurring solution often mentioned by participants. This suggests that software documentation does not receive the attention it deserves when planning and allocating software resources. This finding is relevant to software effort estimation models [MJ03] that should explicitly consider the cost of documentation as one of the factors impacting the final effort needed to build a software system.

The low number of participants (28% of them) who indicated *missing code comments* as a major concern was unexpected. Previous studies [dSAdO05] confined the importance of code comments to general software engineering activities. Some practitioners attributed missing comments to understaffed projects, where the team tends to focus more on coding rather than on documenting. However, they highlighted the need for *"instilling discipline into the team and encouraging writing code comments as a good coding practice"*. Practitioners also indicated the need for automation in code comment generation, *e.g.,* generating templates for bootstrapping the writing process and/or automatically generating (part of) code comments. The latter is an active research area [MM14, MLMW14, MM16, LVLVP16, LVLV+16, LVLVP18] and a roadmap has been proposed by Robillard *et al.* [RMT+17].

The results of our survey confirm the potential and need of automated documentation generation techniques. An exception was an answer provided by a practitioner advocating for writing code in a *"self-explanatory way, with as few comments as possible"*.

Automated tools are also invoked by practitioners to address issues caused by *missing diagrams*. One of them suggested that *"it should be easier to create graphs/diagrams from the text"*.

While approaches for generating diagrams from low-level artifacts (*e.g.,* source code) exist (see, *e.g.,* [KPVDBM06]), practitioners call for approaches that support the extraction of diagrams (*e.g.,* components diagrams) from high-level text-based artifacts (*e.g.,* requirements). The development of these techniques poses interesting research challenges, such as the identification of components needed to implement a given requirement as well as their interactions. This represents an interesting direction for future research.

Regarding *Up-to-dateness* issues, the *lack of documentation for a new feature/component* was not only the one considered important by most participants (69% of them) but also the most recurring issue in this subcategory. Practitioners tend to resort to external sources of information to understand the new feature, or to contact the appropriate parties (*i.e.,* the team who developed the feature) to ask for explanations. One participant stressed the importance of documenting code implementing new features by using concrete examples: *"some parameters are impossible to understand without documentation, and documentation is often not very good"*. Here, automated documentation tools appear again as a solution to enable automated refactoring of documentation, and up-to-date documentation generation as part of continuous integration pipelines.

Inconsistency between code and documentation was also perceived as an important issue by practitioners (59% of them) and is one of the top recurring issues they face.

This observation is in line with previous studies [CH09, PDS14, UR15], which found documentation consistency to be a major issue. The most common up-to-dateness documentation issues involve code comments, which might not reflect changes implemented in the code [WNBL19].

An interesting observation is related to the maintainability of code comments: One practitioner highlighted that a solution for up-to-dateness issues is to limit code comments to the minimum needed, so it is simpler to co-evolve them with code (*i.e.,* if a comment documents useless details, it is more likely that changes implemented in the code will impact it).

The research community has focused on the generation of code comments, while our survey points to the need for approaches that identify redundant and/or unnecessary code comments that increase the comment maintenance cost. Maintainability of comments is a real concern.

While other *Up-to-dateness* issues were considered important by practitioners (see Figure 6.2), exceptions to this trend were: *outdated license/copyright information*, *outdated screenshot*, *outdated translation*, and *outdated version information*.

Summing up

In the *Information Content (What)* category, 7 out of 23 (30%) documentation issues from our taxonomy [ANVM+19] are perceived as important by the majority (≥60%) of surveyed practitioners (*e.g., faulty tutorial*, *inappropriate installation instructions*, *missing documentation for a new feature/-component*). Instead, nine issues (39%) are considered relevant by less than 40% practitioners—see *e.g., wrong translation*, *outdated screenshot*, *outdated license*. This is a first indication that, despite the comprehensiveness of our taxonomy [ANVM+19], the research community could prioritize selected issues that are actually relevant to practitioners.

6.3.2 Information Content (How)

This category of issues is related to the way documentation content is written and organized. Regarding *Maintainability* issues, practitioners considered *superfluous content* (55% of them) and *clone/duplicate content* (46%) the main sources of concern. This observation is in line with our previous study [ANVM+19], which reports that these two subcategories are responsible for ∼71% of developers' discussions on maintainability of documentation.

Given the frequency of these issues [ANVM+19] and their relevance to practitioners, the research community could leverage existing technologies to provide (even partial) solutions. For example, as code clone detection approaches have been defined in the literature [RCK09], similar techniques using natural language processing could be developed to identify (and suggest how to refactor) cloned content in software documentation.

In the case of *superfluous content*, a compelling next step would be to qualitatively study this type of content to develop techniques for its automatic detection (similarly to the work in code smell detection, with a combination of empirical studies [TPB+17b] and detection techniques [MGDLM10]).

Concerning *Readability*, documentation *clarity* is the issue perceived as most important by practitioners (88% of them), as it affects them in many ways (*e.g., "A developer in our team created confusing and overly complicated documentation for customers of our solution"*, *"We experienced this issue when deploying an app that was built by a third party that no longer supports us; the documentation they provided is not clear on how to configure it properly"*).

Previous studies also reported the importance of *Readability* [PDS14, GGM+13, GGYR+15, PDS14] and *Understandability* [PDS14].

To deal with this issue, a number of solutions were proposed by participants. First, documentation writers should always keep in mind the actual documentation users and their needs when writing a document. Second, documentation should be tested by someone with little domain knowledge. For example, if the documentation at hand is an installation guide, it should be tested by users of the system rather than a development team member. This, according to the practitioners, would help in promoting documentation clarity. Moreover, one participant stressed the importance of selecting essential information items in the documentation, highlighting them, and investing in

their writing. Our second research question investigates the information items, from different types of documentation, that are more useful during specific software engineering activities.

In the *Usability* subcategory, issues related to *accessibility/findability* and *information organization* are considered important by practitioners (65% and 49% of them, respectively), while others (*e.g., excessive website load-time*, *violation of best practices in example code*) are not perceived as such.

These findings are in line with our previous results [ANVM+19]. Other previous works [GGM+13, GGYR+15, PDS14, UR15] have revealed the importance of documentation organization and its impact on content findability, confirming the relevance of these issues.

Studies on how users interact with and search documentation could help the research community to define clear guidelines on how different types of documentation should be organized to address accessibility/findability issues.

Format/presentation issues in the documentation are frequently encountered by practitioners who, however, do not consider them as a major issue (only 33% of the participants perceived them as important). The most common suggested solution for this type of issues is to provide documentation guidelines and standard templates. Tools to validate documentation format and adherence to a predefined template would be useful in this context.

In the *Usefulness* subcategory, 59% of the surveyed practitioners considered *content is not useful in practice* as a major and frequent issue. Reviewing the documentation before release, and providing more in-depth details and practical examples were suggested solutions to deal with this issue. The prevalence of this usefulness issue stresses further more the importance of our second research question, *i.e.,* knowing the information items that are actually useful for practitioners can help in avoiding useless content.

Summing up

In the *Information Content (How)* category, only 2 out of 12 (17%) documentation issues from our taxonomy [ANVM$^+$19] are perceived as important by at least 60% of surveyed practitioners (*i.e., clarity* and *accessibility/findability*).

6.3.3 Process/Tool Related

Compared to the previous two categories, developers found issues related to the documentation process and tools less important (see Figure 6.2). One substantial difference between this category and the previous two is that, as indicated by participants, issues in this category mostly affect documentation writers, while both writers and readers are affected by most of the issues in the other categories.

Lack of time to write documentation is the only issue in this category that was indicated as important by the majority (65%) of participants. Also, it is a frequently encountered issue. The proposed solutions boil down to: (i) explicitly allocating time/effort/resources to documentation in the project planning; and (ii) starting documentation activities in the early stage of the software lifecycle, to avoid situations such as the one described by a practitioner: *"The documentation team comes into picture only at the last moment before the release"*. The consequences of inadequate documentation planning were also stressed by another practitioner: *"Projects were built without providing documentation; as time passes aspects of the project are forgotten which makes revisiting the project when updates or modifications need to be made more difficult"*. As highlighted by a respondent, *"when estimating time to complete a project, it is important to make sure that documentation is counted in"*.

Among other *Process/Tool Related* issues, poor *organization of documentation files* and *traceability* issues were frequently encountered by developers, even though only 40% and 35% of them, respectively, considered these issues important.

Lack of traceability in documentation has also been reported in previous work [CH09]. This issue could be addressed by investing in professional tools that integrate traceability recovery techniques proposed in the literature [ACC$^+$02].

Regarding *Documentation tools* issues, the *lack of/poor automatization* was the only issue frequently faced by practitioners. Smarter tools, better IDE integration, and automated generation of documentation were the common requests in this context.

The need for automation can be justified by the *lack of time* issue discussed above. The research community is already investigating novel techniques for the automatic generation of documentation [RMT$^+$17], and our survey confirms the practical relevance of this research area to practitioners.

Participants mentioned other tool-related issues, such as the lack of training for teams or the lack of good tool support for some languages: *"Writing good docs for C/C++ is hard; there are no tools that capture function/class semantics [...]; this would allow the automation of at least a part of the doc writing process"*.

Summing up

Concerning the *Process/Tool Related* category, the majority of issues in our taxonomy [ANVM$^+$19] are not considered important. The notable exception is represented by the *lack of time to write documentation*. Four additional issues are frequently faced by practitioners, including the *lack of/poor automatization*.

6.4 What types of documentation are useful to practitioners?

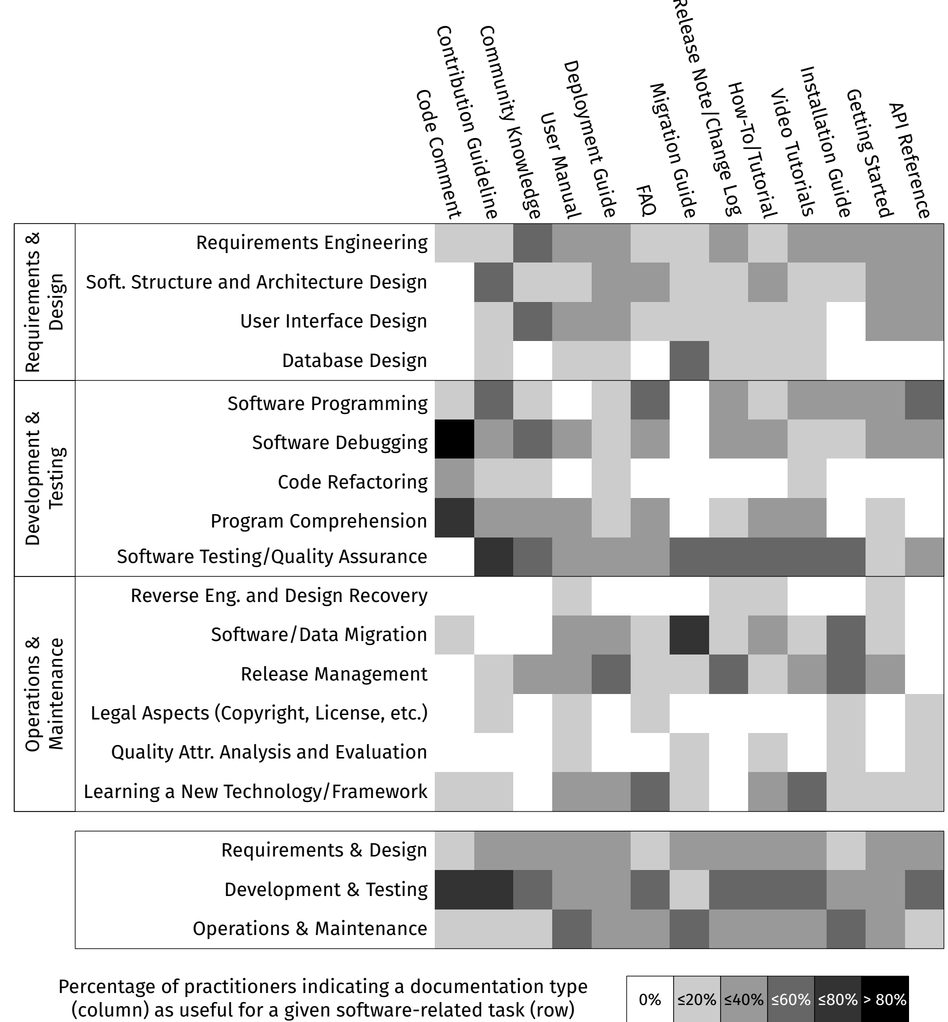


Figure 6.3. Types of documentation perceived as useful by practitioners in the context of specific software engineering tasks (RQ_2), according to the results of Survey-II

Figure 6.3 presents in a heat map the percentage of practitioners indicating documentation types (columns) as useful for given software engineering tasks (rows). A dark spot represents an artifact that was found to be particularly useful for a specific task (*e.g., Code Comment* for *Software Debugging*), while a white spot shows that none of the participants considered the documentation artifact useful for a given task (*e.g., Code Comment* for *Database Design*).

We grouped and sorted the software engineering tasks based on the stage of the software lifecycle to which they relate the most. This resulted in three groups of tasks: *Requirements & Design*, *Development & Testing*, and *Operation & Maintenance*. We present a heat map that combines the data about the different tasks in the mentioned groups at the bottom of Figure 6.3.

The documentation types are sorted based on the average percentage of participants who considered the documentation type useful for each of the 15 tasks.

In the following, we discuss interesting cases, while more detailed results are available in the

replication package.

Code Comment and *Contribution Guideline* were the two documentation types considered as more useful for the different tasks. The distributions of answers are quite different and skewed towards different tasks. *Code Comment* was found highly useful for only a few *Development & Testing* tasks. In particular, all participants agreed on the usefulness of *Code Comment*s for *Software Debugging* and 80% of them also pointed to their importance for *Program Comprehension*. 40% of practitioners marked *Code Refactoring* as an activity benefiting from code comments. Concerning other tasks, excluding isolated exceptions, participants did not mark code comments as useful.

When asked about why *Code Comment*s are useful for the aforementioned tasks, participants emphasized that comments communicate information that is not evident in the code but could help other developers, particularly those who join the project at later stages: *"Software is developed over a long period of time by many different developers; something that may seem obvious to one person may not be obvious to the person who has to maintain the code"*. Concerning the useful information items from comments, participants who marked them as useful for debugging highlighted the fundamental role of parameters' descriptions, *e.g., "Comments might tell the meaning of parameter if the variable name is not adequate"*. Assumptions made in the code should be also documented since they serve in debugging. Practitioners mostly see comments as a way to gather information about the code purpose and behavior.

The different information items in code comments deemed useful for two quite related tasks (*i.e.,* code debugging and program comprehension) confirms that the context is essential in documentation recommender systems aimed at automatically generate documentation (*e.g.,* comments) or at pointing to useful sources of information [Bav16, RMT+17]. More in general, it highlights the importance of keeping comments updated and consistent with the source code, as also observed in our previous study [ANVM+19].

Only a few approaches, however, are available to detect code-comment inconsistencies, but they are specialized to specific types of comments (*e.g.,* Javadoc [TMTL12]).

Contribution Guideline was found helpful in development tasks (64% of the participants found it practical for *Software Testing/Quality Assurance*, and 45% for *Software Programming*), but it is more versatile than *Code Comment* and suitable for design tasks too: 45% of participants claimed its usefulness during *Software Structure and Architecture Design*. They mentioned various benefits derived from the usage of this documentation, *e.g.,* consistent style and use of common programming techniques, improved productivity of new contributors, explanation of workflows, CI pipelines, and releases processes. Interestingly, a sales manager wrote about the importance of this documentation as a means to demonstrate to the customer the quality of the developed products: *"Specifically I deal with pre-sales and eventually end user. This document would validate our concerns and investments in the product [...], so it is very important"*. Looking at the useful information items from this document, practitioners mostly mentioned *best practices*, *coding style guidelines*, *testing requirements*, and *pull request/release checklist*. The last two were considered as particularly important for *Software Testing/Quality Assurance*.

Contribution Guideline is an often neglected document that does not seem to have a negative effect when it is missing but, as shown by our survey, can positively influence a wide range of tasks when it is well written. It is also poorly considered in the researcher community; hence, there are many open possibilities to help practitioners. An interesting direction could be to (partially) generate such a document by automatically recognizing development practices, *e.g.,* coding styles, testing practices, frequently reviewed aspects in pull requests. Such a tool could benefit from learning approaches (*e.g.,* Allamanis *et al.* [ABBS14]).

Another documentation type useful for many tasks is the *User Manual*. It was found helpful for most of the tasks by at least one fifth of the participants. The most important information items from this document are, according to the surveyed participants: *Screenshots*, *Description of expected behavior*, and *Step-by-step technical descriptions*. However, also in this case, we observed differences in the distribution of these items depending on the task. For example, *Screenshots* and *Description of expected behavior* are useful for *Software Debugging* (to check how the system should behave).

Despite the apparent relevance of a *User Manual* in many tasks, our taxonomy [ANVM+19] has only one issue (*Missing User Manual*) directly mentioning it, although other more general issues in the taxonomy could apply to this type of document. Given the recognized importance of *User Manuals*, researchers have started working on approaches to automatically generate/update parts of this document, *e.g.,* screenshots. Representative examples are the work by Waits and Yankel [WY14] and Souza and Oliveira [SO17].

There are several white spots in Figure 6.3, partially due to the low number of responses we collected for *Code Refactoring*, *Reverse Engineering and Design Recovery*, *Legal Aspects*, and *Quality Attributes Analysis and Evaluation*. Recall that participants could choose up to three tasks for each assigned documentation type, so it is possible that they did not select other tasks for which the given documentation might still be useful, but not as much as for the selected top-3. On average, participants selected 2.6 tasks per documentation item, so they indeed took the opportunity to select three tasks in some cases. This could explain the limited number of feedback for some tasks. It also explains why *API Reference* was found helpful in fewer cases than one might expect. Instead of selecting *Code Refactoring* or *Code Comprehension* development tasks, participants put their priorities on other tasks for this type of documentation.

For *API Reference*, most participants mentioned *Overviews/Summaries of fields/methods* and *Code Examples* as relevant, in line with our previous findings [ANVM+19], where we also found many issues related to code examples, and stressed the importance of ensuring the consistency of code examples and the actual code. It also supports the importance of research fields such as code summarization [MM14, MM16, HAM10, HAMM10] and the automatic generation of code examples [MBDP+15].

Finally, the perceived usefulness of *How-To/Tutorials* for different tasks is noteworthy. This is expected when considering the increasing availability of online tutorials about many different topics.

Summing up

The main message resulting from RQ_2 is that practitioners perceive different documentation types as useful for different tasks. Also, as shown in some of the discussed examples, even within the same documentation type, different information items might be useful for different tasks. This supports the need for context-aware documentation recommender systems [Bav16, RMT+17].

6.5 Threats to Validity

Construct validity. They are primarily related to the process we used to select the types of documentation issue (*Survey-I*) and the list of documentation types and tasks (*Survey-II*) for our surveys. These selections may not be representative of all possible documentation issues/types and software-related tasks. To mitigate this threat we always included a free-form “Other” option in the set of answers where these lists were used.

Internal validity. One factor is the response rate: while it does not look very high (9.5%), it is in line with the suggested minimum response rate for survey studies, *i.e.,* 10% [GJC+09].

Another possible threat concerns the fact that 146 respondents decided to participate to the survey because they had greater interest in documentation than others, thus providing a "biased view" of the investigated phenomena. However, our population consists of practitioners that have different roles and, as shown by the results, quite different views on the documentation issues and on the usefulness of different types of documentation for specific tasks.

Finally, a typical co-factor in survey studies is the respondent fatigue bias. We mitigated this threat by running a pilot study with four professional developers and four PhD students, to make sure that both surveys could be answered within 15 minutes.

External validity. The obvious threats here are (i) the context of our study, limited to participants mostly from a single multinational company, and (ii) the total number of participants (*i.e.,* 146). Concerning the first point, while developers from different companies/domains could have different views of the studied phenomena, we collected answers from 20 different countries across 4 continents (complete data in our replication package). As for the number of participants, it is higher or in line with many previously published survey studies [FL02, KM05, dSAdO05, Rob09, DR10, GGM+13, PDS14, SMAR17].

6.6 Summary and Conclusion

We presented two surveys conducted with a total of 146 practitioners. The first survey (*Survey-I*) aimed at investigating documentation issues they perceive as important, and possible solutions they adopt when facing these issues. The second one (*Survey-II*) investigated documentation types they perceive as useful during specific software engineering tasks.

For *Survey-I*, we started from our previous taxonomy [ANVM+19]. Said taxonomy had not been validated with practitioners, and indeed our first study showed that only a small subset of the 162 documentation issues reported in our taxonomy are deemed important by practitioners. Based on the survey responses, we provide a set of suggestions (⚗) for future research endeavours, some of which are surprisingly low hanging fruits. In essence, it does not take much to ameliorate the state of affairs around documentation, and we believe that our now validated taxonomy represents a good starting point.

As for *Survey-II*, our results show when practitioners deem certain documentation types more important for specific tasks at hand. As the research community is headed towards the development of automated documentation generation recommenders [Bav16, RMT+17], we believe that our second study provides precious knowledge for the road ahead.

Part III

Epilogue

*In **Part II** we described our main research accomplishments with respect to software documentation issues. In Chapter 3 we introduced our first study on the effect of poor documentation of software systems. In Chapter 4, we presented* ADANA*, a novel approach for automating documentation. We also discussed its limitations and highlighted the need for empirical studies in this context. In Chapter 5, we presented our first large-scale empirical study on software documentation issues which led to a detailed taxonomy of 162 types of issues faced by developers and users of software documentation. Finally, we extended our previous study in Chapter 6, and presented two surveys with practitioners to learn more about the significance of documentation issues, and the types of information which are more important to developers.*

*In **this part**, in Chapter 7, we conclude this work by reiterating the contributions we made in the course of this thesis and by describing steps we envision as future work.*

7

Conclusions and Future Work

During the course of this dissertation we presented our research achievements towards automating developer documentation by (1) studying the nature of software documentation with a specific focus on documentation issues experienced by software developers, and (2) developing a recommender system supporting the code comprehension process.

In the former direction, we conducted empirical studies and surveys to better understand the nature of software documentation, as well as practitioners' needs in this context. As a result, we have found extensive empirical knowledge, valuable to both researchers in designing the next generation of documentation recommender systems, and developers to improve their projects' documentation-related activities. Moreover, we developed a number of tools and framework with the goal of supporting developers with program comprehension, from line-level code comprehension, to high-level system evolution comprehension.

In this chapter, we sum up all the contributions and findings we discussed in this dissertation, in particular through Chapters 3-6, and outline possible directions for future work.

Structure of the Chapter

- **Section 7.1** highlights our major contributions and accomplishments.
- **Section 7.2** outlines possible future directions for research.
- Finally, **Section 7.3** concludes this chapter.

7.1 Contributions

In this dissertation, we made a series of contributions to the state of the art in software documentation. The contributions of our research can be grouped in two high-level categories: (i) Empirical studies on software documentation, and (ii) Supporting tools and frameworks. In the following, we summarize the major ones.

7.1.1 Empirical studies

The major part of this dissertation is devoted to our empirical studies. We conducted empirical studies and surveys to better understand the nature of software documentation, as well as practitioners' needs in this context. We also investigated software documentation issues and their impact of software systems. These studies can provide the foundations for the next-generation tools and techniques for automated software documentation. In the following we outline these studies:

The impact of poorly documented APIs [ANBL18] With our first study, we studied the impact of poorly documented APIs on software systems. More specifically, the impact of Linguistic Antipatterns (LAs) affecting APIs on (the developers of) client projects using such APIs. For that, we considered 75 popular Maven libraries and their 14k client projects, and we performed a large-scale study to investigate (i) the likelihood of introducing more bugs in the client projects using such APIs, and (ii) whether such APIs trigger more questions on *Stack Overflow*.

Our statistical analysis indicated that when introducing for the first time APIs affected by LAs in the code base, developers have 29% higher chance of introducing bugs as compared to when using clean APIs. In our qualitative analysis, nevertheless, we did not find any strong evidence of their negative impact on the likelihood of introducing bugs. Likewise, we did not find clear evidence that the existence of LAs admittedly triggers questions on *Stack Overflow*. We notice, however, that some LAs are more likely to induce code misunderstandings.

A Large-scale Empirical Study on Linguistic Antipatterns Affecting APIs [ANBL18]
Emad Aghajani, Csaba Nagy, Gabriele Bavota, Michele Lanza
In *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*, pp. 25–35. IEEE, 2018.

Software documentation issues [ANBL18] We presented a large-scale empirical study where we mined, analyzed, and categorized 878 artifacts from four different sources to derive a detailed taxonomy of 162 types of issues faced by developers and users of software documentation (see Figure 5.1). The taxonomy hierarchically organizes the issues into four main categories.

We qualitatively discussed our findings and expose implications for developers and researchers, with the goal of highlighting good practices and interesting research avenues in software documentation.

Software Documentation Issues Unveiled [ANVM+19]
Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza
In *Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*, pp. 1199–1210. IEEE, 2019.

Practitioners' view on software documentation [ANLV+20] Given the large number of issues found in the previous study, and towards our goal of automatically documenting generating documentation, we decided to identify issues relevant to developers. For that, we

conducted two follow-up surveys performed with 146 practitioners to investigate (i) the documentation issues they perceive as more relevant together with solutions they apply when these issues arise; and (ii) the types of documentation considered as important in different tasks. As previously stated, such empirical knowledge can direct future research for designing the next generation of documentation recommender systems.

Software Documentation: The Practitioners' Perspective [ANLV+20]
Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, David C. Shepherd
In *Proceedings of 42nd ACM/IEEE International Conference on Software Engineering (ICSE 2020)*, 12 pages, To be published. IEEE, 2020.

7.1.2 Supporting tools & frameworks

In order to help developers with program comprehension activities we devised techniques and approaches, outlined in the following:

ADANA framework [ABLVL19] Towards our goal of automatically documenting source code, we came up with the idea of ADANA, a framework which generates fine-grained code comments for a given piece of code at the granularity level intended by the developer. ADANA is implemented as a framework that includes an Android studio plug-in, a set of backend services for analyzing and extracting data from online repositories, and a knowledge base for storing snippets and descriptions. In a nutshell, ADANA reuses the descriptions of semantically similar and well-documented code snippets in its knowledge base.

ASIA clone detector [ABLVL19] ASIA is a clone detection approach tailored for Android that we have designed to support ADANA. The ADANA framework is powered by a knowledge base of well-documented code snippet, and benefits from ASIA to identify semantically similar code snippets from this knowledge base. Our evaluation has shown that ASIA clone detector can find similar code snippets with high precision (∼70%).

Code Time Machine [AMBL17] The Code Time Machine aims to help system evolution comprehension with the support of visualization. Our tool allows both developers and the system itself to seamlessly move through time and uses visualization techniques to depict the history of any chosen file augmented with information mined from the underlying versioning system. The tool also enables developers to mark two versions and compare their source code to figure out how or why the value of a code metric has changed at some point in time.

The Code Time Machine [AMBL17]
Emad Aghajani, Andrea Mocci, Gabriele Bavota, Michele Lanza
In *Proceedings of 25th IEEE International Conference on Program Comprehension (ICPC 2017)*, pp. 356–359. IEEE, 2017.

7.2 Future Work

This section elaborates possible future research directions that might be derived from this thesis, and also presents our long-term vision of future recommender systems allowing to automatically document software artifacts, *e.g.,* source code.

Section 7.2.1 presents short-term future research directions raised from studies presented in this dissertation, including ideas on how to overcome limitations of our approaches. After that, Section 7.2.2 presents the long-term vision, focused on the building a context-aware proactive recommender system supporting developers with code comprehension activities.

7.2.1 Short-term future work

1. **The impact of poorly documented APIs** [ANBL18]
In Chapter 3 we presented our study on the impact of Linguistic Antipatterns (LAs) affecting APIs on client projects. As we concluded in Section 3.6, our statistical analysis indicated some effect of LAs on the likelihood of introducing bugs and of triggering Stack Overflow questions, though our qualitative analysis did not allow us to explain such a phenomenon.

   Clearly, this does not contradict the strong empirical evidence showing the negative impact of LAs on code comprehensibility [ADPA16, FMAA18], nor the fact that LAs are considered as bad programming practices by software developers [ADPA16]. However, our findings call for additional investigation about the impact of LAs on code-related activities, maybe conducted through controlled experiments better allowing to isolate the effect of the studied variable.

2. **Automatically generating documentation** [ABLVL19]
In Chapter 4 we introduced ADANA, a novel approach which automatically generates fine-grained code comments for a given piece of Android-related code at the granularity level intended by the developer. ADANA provides a first basic implementation of the ideal recommender system we would like to develop in the long-term run.

   While the results achieved in the performed evaluation are already encouraging, it has strong limitations. For example, it relies on limited contextual information (only the code in the IDE), can only be invoked on-demand (*i.e.,* it is not proactive), and has limitations for what concerns the granularity (*e.g.,* it cannot document multiple code elements at once by also considering and describing their relationships). In this regard, we further discuss possible future research directions in Section 7.2.2 where we explain our vision of the future of automated software documentation.

   Moreover, although the achieved results demonstrated that ADANA can boost the program comprehension process by generating comments which were mostly considered as “useful” by the study participants, we believe that the strength of ADANA lies in the always increasing amount of data that it will be able to exploit in the mined online resources, making ADANA better and better over time. Future work can be devoted to enlarging the ADANA knowledge base. This will be mainly done by defining techniques able to identify well-commented code snippets in open-source software repositories (*e.g.,* by mining the change history to identify commits in which a snippet of code and its comments are added to the system), thus dramatically increasing the amount of information in our knowledge base. As a long-term vision the tool should be able to exploit as much crowdsourced knowledge as possible to automatically document software systems.

   Besides, another potential future research track could be dedicated to enlarging the ADANA support to Java applications in general.

3. **Empirical studies on software documentation** [ANVM+19, ANLV+20]
Chapter 5 and Chapter 6 were devoted to understand the nature of software documentation. For that, we conducted a large-scale study on issues faced by developers and users of software documentation, leading to a detailed taxonomy of 162 types of documentation issues. Following that, we performed two surveys with the goal of identifying practitioners’ needs in this context.

   Throughout these two chapters, we extensively discussed our findings and their implications in software research and practice, deriving actionable items needed to address them. For instance, in the “Discussion and Implications” of page 82, we bring up the correctness issue of

code examples in documentation. For that, we call for future research on devising approaches to test code examples in documentation and validate the consistency between snippets and source code. We invite interested readers to refer to Section 5.3, Section 6.3 and Section 6.4.

Needless to say, to achieve high-quality documentation, no matter by what means, we require first a deep understanding of documentation characteristics and developers needs. Therefore, we strongly encourage researchers and practitioners to invest further effort in this research area.

7.2.2 Our long-term vision

Our primary goal, in the long-term, is to build a context-aware proactive recommender system supporting the code comprehension process. The system must be able to understand the context, consider the developer's profile, and help her by generating pieces of documentation at whatever granularity is required, *e.g.,* going from summarizing the responsibilities implemented in a subsystem, to explaining how two classes collaborate to implement a functionality, down to documenting a single line of code. Generated documentation will be tailored for the current context (*e.g.,* the task at hand, the developer's background knowledge, the history of interactions).

In Chapter 4 we presented our first steps toward our goal by introducing the ADANA project, a framework which generates fine-grained code comments for a given piece of code. Still, it has strong limitations. To address these issues and achieve our goal of generating high-quality documentation, we target the following characteristics for our future recommender system:

1. **Information granularity and presentation.** The recommender system must be able to present information at different granularity levels. For instance, facing a class to comprehend, one developer might be interested only in a high-level description, while others might expect instantiation examples. Thus the system must be able to generate documentation at different granularity levels and provide the developer with a "show me more/fewer details" mechanism. Also, the way the documentation is presented to the developer is important. In particular, we will investigate different presentation techniques to select the most appropriate one based on the amount and type of data to present.

2. **Code granularity.** As we discussed in Section 2.2, most of the state-of-the-art approaches work at a fixed granularity level (*e.g.,* method level). However, the recommender system we have in mind must be able to produce pieces of documentation at whatever granularity is required, *e.g.,* going from summarizing the responsibilities implemented in a subsystem, to explaining how two classes collaborate to implement a functionality, down to documenting a single line of code.

   ADANA, to some extent, fulfills this objective by allowing a developer to choose the intended granularity level (see Figure 4.2). However, ADANA is not able to generate coarse-grained comments (*e.g.,* describing a class or subsystem).

3. **Self-improving.** We want to explore the usage of feedback mechanisms to allow our techniques to self-improve over time. For example, once a piece of documentation is automatically generated for a given code snippet, the developer can provide feedback indicating whether it was useful or not to comprehend the code. This feedback can be then exploited to infer good and bad "commenting patterns", possibly customized on the basis of the developer's preferences. As explained in Section 4.2.1, with ADANA, we partially addressed this objective.

4. **Context-aware.** Currently, recommender systems supporting developers during code-related activities exploit the code in the IDE as the main source of information to capture the context, then recommend useful pieces of information for it (*e.g.,* related Stack Overflow discussions). However, the working context is much more than a bunch of lines of code shown in the IDE. For example, two developers having diverse expertise level should receive different types of documentation even when working on the same task on the same code component. Ignoring the developers' profile might result in generating documentation that is either "too trivial" or "too complex", thus useless in both cases. We plan to capture the context in which documentation will be generated by considering:

   - **Developer's profile.** Information such as the code fragments developed in the past, the developer's expertise on the different technologies used in the project, and the history of successful and unsuccessful tasks performed in the past from a developer's profile. For example, a bug fixed by a developer and not "reopened" in the future can be considered a successful task performed by the developer, while a fixed bug that has been then reopened and fixed again, can represent an instance of an unsuccessful task. Knowing this can help in assessing the experience level of the developer on a particular subsystem (*i.e.,* the one involved in the bug-fixing activity), thus, to tailor the generated documentation to the specific developer's profile.
   - **Task at hand.** Considering the developer's task at hand can provide useful hints to narrow the type of information needed. For example, during a bug-fixing activity it is important to understand, thus, to automatically document, the production code as well as the test code. The idea of utilizing the current task information to assist developers has been adopted and shown to be effective [KM06, HM08] and has been implemented in Mylyn[1], a plug-in for the Eclipse IDE. However, this information must be explicitly indicated by the developer, and it is not automatically inferred by the tool. We plan to exploit data from the issue tracking system to infer the tasks the developer is working on.

   In order to utilize such contextual information, we also need to understand what type of information developers need in different scenarios (*e.g.,* bug fixing vs. implementing a new feature). Our second survey (*Survey-II*) in Chapter 6 goes in this direction.

5. **Heterogeneous sources of information.** As previously discussed, the automatic documentation of source code can be achieved in different ways (*e.g.,* extractive *vs.* abstractive summaries). What clearly makes a difference in the ability to document a given code are the basic information sources exploited. We plan to exploit not only the official project's documentation, but also the project's repositories (*e.g.,* history of changes, information extracted from the issue tracker), as well as the information that can be mined from the Web such as documentation written by other developers for code similar to the one to be automatically documented. With ADANA, we also partially addressed this goal as it exploits two sources of information which are completely different in terms of data storage and presentation.

6. **Proactive and On-Demand.** While we want the developer to be able to interact with our system on-demand (*i.e.,* by asking explicit questions), we also want to provide proactive recommendations in case, for example, we can infer that the developer is struggling to understand a code snippet. This can be done by monitoring her behavior in the IDE and observing patterns likely indicating understandability issues (*e.g.,* scrolling up and down several times over

[1] See `http://www.eclipse.org/mylyn/`

a specific method). In this cases, the tool could automatically document the code with hints on what the different lines of code implement. Concerning the on-demand invocation, we plan to guide developers in expressing their information needs.

7. **Documentation Presentation and Navigation** The way developers seek information through documentation comes under the direct influence of how the documentation is presented. Early studies by Sheppard *et al.* [SKB82, CSKB+89] have revealed that documentation format affects developers performance. Despite this, little research effort has been invested in devising techniques to present software documentation in a meaningful and effective way, and our knowledge is mostly limited to best practices discussed in documentation guidelines [goo20, jav20, Rüp05] and books [Els98].

   Bayer and Muthig [BM06] revealed that the quality of the documentation affects its usage and could be improved by taking the documentation end uses into account when the documentation is being created. In this regard, a qualitative survey on information management in software development by Olsson and Runeson [OR] has shown that natural language is the predominant notation in software documentation.

   Future work in this context can be devoted to exploring the use of different presentation techniques (*e.g.,* text, table, diagram) and studying their impact on developer's comprehension. Additionally, learning about type of visualization that works best for each type of information could be another direction for future work.

7.3 Closing Words

Our journey started with one goal in mind: enhancing the code comprehension activity and promoting software documentation as a first-class citizen in software systems. In this thesis, we presented a series of empirical studies on the nature of software documentation. Our contributions to the body of software documentation knowledge shed light on unseen facts about overlooked software documentation matter and lay the foundations for the next-generation tools and techniques for automated generation of software documentation.

At the end, we hope more researchers and practitioners will accept the fundamental notion that documentation is not a mere add-on to any software system, but a part of the system itself.

Bibliography

[10b17] https://github.com/ashutoshgngwr/10-bitClockWidget, 2017.

[AAHMA16] Nouh Alhindawi, Obaida M Al-Hazaimeh, Rami Malkawi, and Jamal Alsakran. A Topic Modeling Based Solution for Confirming Software Documentation Quality. *Int. Journal of Advanced Computer Science and Applications*, 7(2):200–206, 2016.

[ABBS14] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, pages 281–293. ACM, 2014.

[ABF+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *Proc. of the 2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, 2017.

[ABLVL19] Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, and Michele Lanza. Automated documentation of Android apps. *IEEE Transactions on Software Engineering*, 45(1):1–13, 2019.

[ACC+02] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[ada19a] ADANA Android Studio plugin (source code). https://github.com/emadpres/adana/, 2019.

[ada19b] ADANA website. https://adana.si.usi.ch/, 2019.

[ada19c] Replication Package for "Automated Documentation of Android Apps" paper. https://github.com/ADANAPaper/replication_package, 2019.

[ADG17] https://developer.android.com/samples, 2017.

[ADPA16] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[Agh18] Emad Aghajani. Context-aware software documentation. In *Proceedings of ICSME 2018 (34th IEEE International Conference on Software Maintenance and Evolution*, pages 727–731. IEEE Press, 2018.

[AMBL17] Emad Aghajani, Andrea Mocci, Gabriele Bavota, and Michele Lanza. The Code Time Machine. In *Proceedings of ICPC 2017 (25th IEEE International Conference on Program Comprehension)*, pages 356–359. IEEE Press, 2017.

[ANBL18] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In *Proceedings of ICSME 2018 (34th IEEE International Conference on Software Maintenance and Evolution*. IEEE Press, 2018.

[and17a] `https://developer.android.com/`, 2017.

[and17b] `https://developer.android.com/reference/packages.html`, 2017.

[ANLV+20] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: The practitioners' perspective. In *Proceedings of ICSE 2020 (42nd ACM/IEEE International Conference on Software Engineering)*, page To be published, 2020.

[ANN+17] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. A systematic evaluation of API-misuse detectors. *arXiv preprint arXiv:1712.00242*, 2017.

[ANVM+19] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *Proceedings of ICSE 2019 (41st ACM/IEEE International Conference on Software Engineering)*, pages 1199–1210. IEEE Press, 2019.

[APAG13] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 187–196, 2013.

[ARG17] Shams Azad, Peter C Rigby, and Latifa Guerrouj. Generating API call rules from version history and Stack Overflow posts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(4):29, 2017.

[AS92] James D. Arthur and K. Todd Stevens. Document quality indicators: A framework for assessing documentation adequacy. *Journal of Software Maintenance: Research and Practice*, 4(3):129–142, 1992.

[AT15] Surafel Lemma Abebe and Paolo Tonella. Extraction of domain concepts from the source code. *Science of Computer Programming*, 98:680–706, 2015.

[Bav16] Gabriele Bavota. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 1–12. IEEE, 2016.

[BEBTM08] Florian Boudin, Marc El-Bèze, and Juan-Manuel Torres-Moreno. A scalable mmr approach to sentence scoring for multi-document update summarization. In *Coling 2008: Companion volume: Posters*, pages 23–26, 2008.

[BF14] Pierre Bourque and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014.

[BH11] Håkan Burden and Rogardt Heldal. Natural language generation from class diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–8, 2011.

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[BLH+13] Dave Binkley, Dawn Lawrie, Emily Hill, Janet Burge, Ian Harris, Regina Hebig, Oliver Keszocze, Karl Reed, and John Slankas. Task-driven software summarization. In *2013 IEEE International Conference on Software Maintenance*, pages 432–435. IEEE, 2013.

[Blo18] Joshua Bloch. *Effective Java (3rd Edition) (The Java Series)*. Pearson Education Inc., 2018.

[BM06] Joachim Bayer and Dirk Muthig. A view-based approach for improving software documentation practices. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 10–pp. IEEE, 2006.

[BOL10] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166. ACM, 2010.

[BTH14] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Softw. Eng.*, 19(3):619–654, jun, 2014.

[BW08] Raymond PL Buse and Westley R Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 273–282, 2008.

[BW12] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792. IEEE Press, 2012.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[Cau10] Andrew H Caudwell. Gource: visualizing software version control history. In *Proceedings of OOPSLA 2010 (25th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 73–74. ACM, 2010.

[CCLVAP14] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284, Sept 2014.

[CH09] Jie Cherng Chen and Sun Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.

[Con98] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 1998.

[CSKB+89] Bill Curtis, Sylvia B Sheppard, Elizabeth Kruesi-Bailey, John Bailey, and Deborah A Boehm-Davis. Experimental evaluation of software documentation formats. *Journal of Systems and Software*, 9(2):167–207, 1989.

[Dau11] A Dautovic. Automatic assessment of software documentation quality. In *2011 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, pages 665–669, nov 2011.

[DCM06] N. Dragan, M. L. Collard, and J. I. Maletic. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 24–34, Sept 2006.

[DCM10] N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[DER12] Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proc. of the 34th Int. Conf. on Software Engineering*, pages 266–276, 2012.

[DH09] Uri Dekel and James D Herbsleb. Improving API documentation usability with knowledge pushing. In *Proc. of the IEEE 31st Int. Conf. on Software Engineering, ICSE 2009*, pages 320–330, 2009.

[doc19] Replication Package for "Software Documentation Issues Unveiled" paper. `https://github.com/REVEAL-ICSE19-DocIssues/ReplicationPackage`, 2019.

[doc20] Replication package for "Software Documentation: The Practitioners' Perspective" paper. `https://github.com/USI-INF-Software/Conf-ReplicationPackage-ICSE2020`, 2020.

[DR10] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. pages 127–136, 2010.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, SIGDOC '05, pages 68–75. ACM, 2005.

[DSPA+16] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. What would users change in my app? Summarizing app reviews for recommending software changes. In *Proc. of the 24th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE 2016)*, pages 499–510. ACM, 2016.

[Els98] Arthur G Elser. *Writing software documentation: A task-oriented approach*, volume 45. Society for Technical Communication, 1998.

[ER04] Günes Erkan and Dragomir R Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of artificial intelligence research*, 22:457–479, 2004.

[ERKC13] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 13–22. IEEE, 2013.

[ESM07] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proc. of the 29th Int. Conf. on Software Engineering*, pages 302–312, 2007.

[fCMA12] Association for Computing Machinery (ACM). The 2012 acm computing classification system, 2012.

[FCR+17] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton. Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12):1095–1109, 2017.

[FL02] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *DocEng'02*, pages 26–33, 2002.

[Fle48] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221, 1948.

[FMAA18] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *ICPC '18: 26th IEEE/ACM Int. Conf. on Program Comprehension*, 2018.

[for17] `https://github.com/romannurik/FORMWatchFace`, 2017.

[Fou] Apache Software Foundation. Apache Mail Archives. `http://mail-archives.apache.org/mod_mbox/`.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, pages 23–, 2003.

[FWG07] Beat Fluri, Michael Wursch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.

[FWGG09] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, dec, 2009.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Software Maintenance and Evolution: Research and Practice*, 18:207–236, 2006.

[GGM+13] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating usage and quality of technical software documentation: an empirical study. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 24–35, 2013.

[GGYR+15] Golara Garousi, Vahid Garousi-Yusifoglu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57(1):664–682, 2015.

[Gis17] `https://gist.github.com`, 2017.

[Git] GitHub. Event Types & Payloads. `https://developer.github.com/v3/activity/events/types/`.

[GJC+09] Robert M. Groves, Floyd J. Fowler Jr., Mick P. Couper, James M. Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey Methodology, 2nd edition*. Wiley, 2009.

[GK05] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2005.

[goo20] `https://developers.google.com/style/`, 2020.

[Gri] Ilya Grigorik. GitHub Archive. `https://www.githubarchive.org`.

[GTGPT11] Antonio González-Torres, Francisco J García-Peñalvo, and Roberto Therón. A framework for the evolutionary visual software analytics process. In *Proceedings of WSKS 2011 (4th World Summit on the Knowledge Society)*, pages 439–447. Springer, 2011.

[GZHK18] Elena Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API usage examples at scale. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, 2018.

[HAM10] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE'10)*, volume 2, pages 223–226. IEEE, 2010.

[HAMM10] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, Oct 2010.

[Har03] Simon Harris. `http://www.harukizaemon.com/simian/`, 2003.

[HDLL11] Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. Software evolution comprehension: Replay to the rescue. In *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension)*, pages 161–170, 2011.

[Hen07] Michi Henning. API design matters. *Queue*, 5(4):24–36, 2007.

[HLL10] Lile Hattori, Mircea Lungu, and Michele Lanza. Replaying past changes on multi-developer projects. In *Proceedings of IWPSE-EVOL 2010 (Joint 11th International Workshop on Principles of Software Evolution and 5th ERCIM Workshop on Software Evolution)*, pages 13–22, 2010.

[HLX+18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[HM05] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125. ACM, 2005.

[HM08] Hans-Jörg Happel and Walid Maalej. Potentials and challenges of recommendation systems for software development. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 11–15, 2008.

[HZ13] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proc. of the 10th Working Conf. on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 121–130, 2013.

[HZF+12] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and W. Strouila. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *WCRE'12*, pages 83–92, 2012.

[IKCZ16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[Jac01] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[JAM+17] Siyuan Jiang, Ameer Armaly, Collin McMillan, Qiyu Zhi, and Ronald Metoyer. Docio: Documenting api input/output examples. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 364–367. IEEE Press, 2017.

[jav20] `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`, 2020.

[JM17] S. Jiang and C. McMillan. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 320–323, May 2017.

[JMSG07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, 2007.

[KBEL12] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and evolution of software architectures. In *Proceedings of IRTG 1131 Workshop 2011, VLUDS 2011 (2nd Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering)*, volume 27 of *OpenAccess Series in Informatics (OASIcs)*, pages 25–42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

[KFRC75] J.P. Kincaid, R.P.Jr. Fishburne, R.L. Rogers, and B.S. Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, 1975.

[KJM17] R. Krasniqi, S. Jiang, and C. McMillan. Tracelab components for generating extractive summaries of user stories. In *2017 IEEE Int. Conf. on Soft. Maint. and Evolution (ICSME)*, pages 658–658, Sept 2017.

[KM05] Mira Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.

[KM06] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[KMA04] Andrew J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 199–206, 2004.

[KMS15] Maria Kechagia, Dimitris Mitropoulos, and Diomidis Spinellis. Charting the API minefield using software telemetry data. *Empirical Software Engineering*, 20(6):1785–1830, Dec 2015.

[KNGW13] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, Jan 2013.

[Kos07] Rainer Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[KPG+13] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar T. Devanbu, and Vladimir Filkov. Using and asking: APIs used in the android market and asked about in StackOverflow. In *Social Informatics - 5th Int. Conf., SocInfo 2013, Kyoto, Japan, November 25-27*, pages 405–418, 2013.

[KPVDBM06] Elena Korshunova, Marija Petkovic, MGJ Van Den Brand, and Mohammad Reza Mousavi. Cpp2xmi: Reverse engineering of uml class, sequence, and activity diagrams from c++ source code. In *2006 13th Working Conference on Reverse Engineering*, pages 297–298. IEEE Comp. Soc., 2006.

[KSNH15] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (4th International Workshop on Principles of Software Evolution)*, pages 37–42. ACM, 2001.

[LD02] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO'02)*, pages 135–149. Lavoisier, 2002.

[LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM, 2005.

[LDS01] Michele Lanza, Stéphane Ducasse, and Lukas Steiger. Understanding software evolution using a flexible query engine. In *Proceedings of FFSE 2001 (1st Workshop on Formal Foundations of Software Evolution)*, 2001.

[LM11] Amy N Langville and Carl D Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton university press, 2011.

[LMC12] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. In *Proc. of the 28th IEEE Int. Conf. on Soft. Maintenance (ICSM)*, pages 430–439, Sept 2012.

[LMC15] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried'bug report reading process to summarize bug reports. *Empirical Software Engineering*, 20(2):516–548, 2015.

[LSF03a] TC Lethbridge, J Singer, and A Forward. Use documentation: The state of the practice documentation. *Ieee Focus*, page 5, 2003.

[LSF03b] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, nov, 2003.

[LSX18] Jing Li, Aixin Sun, and Zhenchang Xing. Learning to answer programming questions with software documentation through social context embedding. *Information Sciences*, 448:36–52, 2018.

[LVCCAP15] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 709–712, May 2015.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006*, pages 492–501. ACM, 2006.

[LVDP13] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR'13)*, pages 93–96, May 2013.

[LVHBCP14] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Working Conference on Mining Software Repositories (MSR'14)*, pages 242–251, 2014.

[LVLV+16] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft. Automatically documenting unit test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352, April 2016.

[LVLVP15] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. How do developers document database usages in source code? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 36–41, Nov 2015.

[LVLVP16] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 270–281. ACM, 2016.

[LVLVP18] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 52–63. ACM, 2018.

[LVMP17] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *ICSME'17*, page to appear, 2017.

[MAS+13] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32, May 2013.

[MBDP+14] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495. ACM, 2014.

[MBDP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, 2015.

[MBP+17] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. Arena: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127, Feb 2017.

[MC15] Jonathan I. Maletic and Michael L. Collard. Exploration, analysis, and manipulation of source code using srcml. In *Proc. of the 37th Int. Conf. on Software Engineering - Volume 2*, ICSE '15, pages 951–952. IEEE Press, 2015.

[MCSD12] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. AUSUM: Approach for unsupervised bug report summarization. In *Proc. of the ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Engineering*, FSE '12, pages 11:1–11:11. ACM, 2012.

[MGDLM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[MJ03] Kjetil Molkken and Magne Jrgensen. A review of surveys on software effort estimation. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, pages 223–230. IEEE Comp. Soc., 2003.

[ML13] Roberto Minelli and Michelle Lanza. Software analytics for mobile applications – insights & lessons learned. In *17th European Conference on Software Maintenance and Reengineering*, page To appear, 2013.

[MLMW14] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In *Proc. of the 22nd Int. Conf. on Program Comprehension (ICPC 2014)*, pages 291–294. ACM, 2014.

[MM14] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 279–290. ACM, 2014.

[MM16] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, Feb 2016.

[MMPVS13] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Jsummarizer: An automatic generator of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 230–232. IEEE, 2013.

[MPG+13] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, oct, 2013.

[MR13] Walid Maalej and Martin P Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[MRNAH12] I.J. Mojica Ruiz, M. Nagappan, B. Adams, and A.E. Hassan. Understanding reuse in the Android market. In *20th IEEE International Conference on Program Comprehension (ICPC'12)*, pages 113–122, 2012.

[MRTS98] Samuel G McLellan, Alvin W Roesler, Joseph T Tempest, and Clay I Spinuzzi. Building more usable APIs. *IEEE software*, 15(3):78–86, 1998.

[MS16] Brad A Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, 2016.

[MSB+14] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'14)*, pages 55–60, 2014.

[MWH06] G. C. Murphy, R. J. Walker, and R. Holmes. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32:952–970, 12 2006.

[NM+11] Ani Nenkova, Kathleen McKeown, et al. Automatic summarization. *Foundations and Trends® in Information Retrieval*, 5(2–3):103–233, 2011.

[OM09] Michael Ogawa and Kwan-Liu Ma. code_swarm: A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104, 2009.

[OM10] Michael Ogawa and Kwan-Liu Ma. Software evolution storylines. In *Proceedings of SOFTVIS 2010 (5th ACM symposium on Software visualization)*, pages 35–42. ACM, 2010.

[OR] Thomas Olsson and Per Runeson. Document use in software development: a qualitative survey. In *Software Engineering, research and practise in Sweden (SERPS'02)*.

[OSA17] `https://github.com/pcqpcq/open-source-android-apps`, 2017.

[PBDP+14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111. ACM, 2014.

[PBL13] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Proc. of the 17th European Conf. on Soft. Maint. and Reeng.*, pages 57–66, March 2013.

[PBM+16a] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. Too long; didn't watch! extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering*, pages 261–272, 2016.

[PBM+16b] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. Codetube: extracting relevant fragments from software development video tutorials. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 645–648. IEEE, 2016.

[PDS14] R Plösch, A Dautovic, and M Saft. The Value of Software Documentation Quality. In *Proc. of the 14th Int. Conf. on Quality Software*, pages 333–342, oct 2014.

[PFM13] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of API usability. In *Proc. of the 2013 ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement*, pages 5–14, 2013.

[PML15] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing complex development artifacts by mining heterogeneous data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 401–405. IEEE, 2015.

[PRDM15] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. Discovering information explaining API types using text classification. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE Int. Conf. on*, volume 1, pages 869–879, 2015.

[PSB+17] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 94–105, 2017.

[PTG12] Chris Parnin, Christoph Treude, and Lars Grammel. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical report, 2012.

[RBK+16] Daniel Rozenberg, Ivan Beschastnikh, Fabian Kosmale, Valerie Poser, Heiko Becker, Marc Palyart, and Gail C Murphy. Comparing repositories visually with repograms. In *Proceedings of MSR 2016 (13th International Conference on Mining Software Repositories)*, pages 109–120. ACM, 2016.

[RC15] Martin P. Robillard and Yam B. Chhetri. Recommending reference api documentation. *Empirical Softw. Engg.*, 20(6):1558–1586, dec, 2015.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, may, 2009.

[RD11] Martin P Robillard and Robert Deline. A field study of API learning obstacles. *Empirical Software Engineering (EMSE)*, 16(6):703–732, 2011.

[Red11] Martin Reddy. *API Design for C++*. Elsevier, 2011.

[res17] https://gist.github.com/MBtech/37f2f3df5dfe5805adfd, 2017.

[RJAM17] Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. Detecting user story information in developer-client conversations to generate extractive summaries. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 49–59. IEEE, 2017.

[RMB11] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 103–112, Sept 2011.

[RMM10] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 505–514. IEEE, 2010.

[RMM14a] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, April 2014.

[RMM+14b] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 390–401. ACM, 2014.

[RMT+17] M P Robillard, A Marcus, C Treude, G Bavota, O Chaparro, N Ernst, M A Gerosa, M Godfrey, M Lanza, M Linares-Vásquez, G C Murphy, L Moreno, D Shepherd, and E Wong. On-demand Developer Documentation. In *Proc. of the 33rd IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pages 479–483, sep 2017.

[Rob09] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[RRK15] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending insightful comments for source code using crowdsourced knowledge. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 81–90, Sept 2015.

[RS16] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.

[Rüp05] Andreas Rüping. *Agile documentation: a pattern guide to producing lightweight documents for software projects*. John Wiley & Sons, 2005.

[SB17] Anand Ashok Sawant and Alberto Bacchelli. fine-GRAPE: fine-grained APi usage extractor–an approach and dataset to investigate API usage. *Empirical Software Engineering*, 22(3):1348–1371, 2017.

[SBV+17] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 417–427. IEEE, IEEE Press, 2017.

[SC07] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proc. of the 29th Int. Conf. on Software Engineering*, pages 529–539, 2007.

[She03] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.

[SHM+10] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori 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*, ASE '10, pages 43–52. ACM, 2010.

[SIH14] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652. ACM, 2014.

[SKB82] Sylvia B Sheppard, Elizabeth Kruesi, and John W Bailey. An empirical evaluation of software documentation formats. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 121–124. ACM, 1982.

[SLOP18] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6), 2018.

[SLVPO16] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[SM06] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding api components and examples. In *Proc. of the Visual Languages and Human-Centric Computing (VLHCC 2006)*, pages 195–202. IEEE, 2006.

[SM08] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proc. of the 16th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 105–112, 2008.

[SMAR17] S. M. Sohan, Frank Maurer, Craig Anslow, and Martin P. Robillard. A study of the effectiveness of usage examples in REST API documentation. *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 2017-October:53–61, 2017.

[SO17] Rodrigo Souza and Allan Oliveira. Guideautomator: Continuous delivery of end user documentation. In *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER*, pages 31–34. IEEE, 2017.

[Spi10] D. Spinellis. Code documentation. *IEEE Software*, 27(4):18–19, July 2010.

[SPVS11a] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.

[SPVS11b] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80. IEEE, 2011.

[Sri12] Giriprasad Sridhara. *Automatic generation of descriptive summary comments for methods in object-oriented programs*. PhD thesis, 2012.

[Sta17] `https://stackoverflow.com`, 2017.

[swi18] `https://swift.org/documentation/api-design-guidelines/`, 2018.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. of the 2005 Int. Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005.

[TM02] Christopher MB Taylor and Malcolm Munro. Revision towers. In *Proceedings of VISSOFT 2002 (1st IEEE International Workshops on Visualizing Software for Understanding and Analysis)*, pages 43–50. IEEE, 2002.

[TMGN16] Yuriy Tymchuk, Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Walls, pillars and beams: A 3d decomposition of quality anomalies. In *Proceedings of VISSOFT 2016 (4th IEEE Working Conference on Software Visualization)*, pages 126–135. IEEE, 2016.

[TMTL12] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269. IEEE, 2012.

[TPB+17a] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2017.

[TPB+17b] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Software Eng.*, 43(11):1063–1088, 2017.

[TR16] Christoph Treude and Martin P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 392–403. ACM, 2016.

[Tul08] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Apress, 2008.

[UR15] Gias Uddin and Martin P. Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, 2015.

[Var16] Ervin Varga. *Creating Maintainable APIs: A Practical, Case-Study Approach*. Apress, 2016.

[VC04] Marcello Visconti and Curtis R. Cook. Assessing the State of Software Documentation Practices. In *Product Focused Software Process Improvement*, pages 485–496, 2004.

[VPDPC14] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. CODES: Mining source code descriptions from developers discussions. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 106–109. ACM, 2014.

[VTVW05] Lucian Voinea, Alex Telea, and Jarke J Van Wijk. Cvsscan: visualization of code evolution. In *Proceedings of SOFTVIS 2005 (2nd ACM Symposium on Software Visualization)*, pages 47–56. ACM, 2005.

[WDZ+13] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proc. of the 10th Working Conf. on Mining Software Repositories*, MSR '13, pages 319–328. IEEE Press, 2013.

[WL08] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.

[WLT15] Edmund Wong, Taiyue Liu, and Lin Tan. CloCom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 380–389, 2015.

[WNBL19] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019*, pages 53–64. IEEE Press, 2019.

[WSJSS13] Robert Watson, Mark Stamnes, Jacob Jeannot-Schroeder, and Jan H Spyridakis. API documentation and software community values: a survey of open-source API documentation. In *Proc. of the 31st ACM Int. Conf. on Design of Communication*, pages 165–174, 2013.

[WTVP16] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, Sept 2016.

[WY14] Todd Waits and Joseph Yankel. Continuous system and user documentation integration. In *2014 IEEE International Professional Communication Conference (IPCC)*, pages 1–5. IEEE, 2014.

[WYT13] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013.

[XP06] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proc. of the 2006 Int. Workshop on Mining Software Repositories*, MSR '06, pages 54–57. ACM, 2006.

[YR13] Annie T. T. Ying and Martin P. Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 655–658. ACM, 2013.

[YR14] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 460–471. ACM, 2014.

[ZGC+17] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs documentation and code to detect directive defects. In *Proc. of the 39th Int. Conf. on Software Engineering*, ICSE '17, pages 27–37. IEEE Press, 2017.

[ZGYS+15] Junji Zhi, Vahid Garousi-Yusifoglu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.

[ZS13] Hao Zhong and Zhendong Su. Detecting API documentation errors. *SIGPLAN Not.*, 48(10):803–816, oct, 2013.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

Artifacts' References

[1] "GitHub Issue of acid-state/acid-state." [Online]. Available: https://github.com/acid-state/acid-state/issues/22/

[2] "GitHub Issue of rockstor/rockstor-core." [Online]. Available: https://github.com/rockstor/rockstor-core/issues/1821/

[3] "GitHub Issue of pluskid/mocha.jl." [Online]. Available: https://github.com/pluskid/Mocha.jl/issues/145/

[4] "GitHub PR of silverstripe/silverstripe-framework." [Online]. Available: https://github.com/silverstripe/silverstripe-framework/pull/5906

[5] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200307.mbox/%3C20030703181821.GA19238@ordiluc.net%3E

[6] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200303.mbox/%3CC1256CE5.005140C6.00@detkw004.dnotes.telekurs.com%3E

[7] "GitHub Issue of tinymce/tinymce." [Online]. Available: https://github.com/tinymce/tinymce/issues/772/

[8] "GitHub Issue of mlpack/mlpack." [Online]. Available: https://github.com/mlpack/mlpack/issues/672/

[9] "GitHub Issue of elliotchance/c2go." [Online]. Available: https://github.com/elliotchance/c2go/issues/278/

[10] "GitHub PR of falconry/falcon." [Online]. Available: https://github.com/falconry/falcon/pull/1086

[11] "GitHub Issue of trilinos/trilinos." [Online]. Available: https://github.com/trilinos/Trilinos/issues/845/

[12] "GitHub Issue of bytedeco/javacpp-presets." [Online]. Available: https://github.com/bytedeco/javacpp-presets/issues/108/

[13] "GitHub PR of payara/payara." [Online]. Available: https://github.com/payara/Payara/pull/1403

[14] "GitHub Issue of nodemcu/nodemcu-firmware." [Online]. Available: https://github.com/nodemcu/nodemcu-firmware/issues/243/

[15] "GitHub Issue of facebook/watchman." [Online]. Available: https://github.com/facebook/watchman/issues/98/

[16] "Apache Mailing List forrest-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/forrest-dev/200304.mbox/%3C20030408042853.62954.qmail@icarus.apache.org%3E

[17] "GitHub PR of coreos/etcd." [Online]. Available: https://github.com/etcd-io/etcd/pull/9265

[18] "GitHub Issue of d3/d3-dispatch." [Online]. Available: https://github.com/d3/d3-dispatch/issues/13/

[19] "GitHub PR of awslabs/aws-sdk-net-samples." [Online]. Available: https://github.com/awslabs/aws-sdk-net-samples/pull/8

[20] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200403.mbox/%3C200403110446.i2B4k1422554@Boron.MeepZor.Com%3E

[21] "GitHub PR of alibaba/rax." [Online]. Available: https://github.com/alibaba/rax/pull/260

[22] "Apache Mailing List systemml-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/systemml-dev/201609.mbox/%3CCAGU5speRbi22jTQB=aWBJ446PbDFG217hDmhF-E5gMSNmm+Y_g@mail.gmail.com%3E

[23] "GitHub Issue of domaindrivendev/swashbuckle." [Online]. Available: https://github.com/domaindrivendev/Swashbuckle/issues/1001/

[24] "GitHub Issue of doctrine/doctrine1." [Online]. Available: https://github.com/doctrine/doctrine1/issues/53/

[25] "GitHub Issue of webpack/docs." [Online]. Available: https://github.com/webpack/docs/issues/75/

[26] "GitHub Issue of stevegrunwell/mcavoy." [Online]. Available: https://github.com/stevegrunwell/mcavoy/issues/18/

[27] "GitHub PR of asciidoctor/asciidoctorj." [Online]. Available: https://github.com/asciidoctor/asciidoctorj/pull/361

[28] "Apache Mailing List cocoon-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/cocoon-docs/200302.mbox/%3C200302042000.h14K02t00326@otsrv1.iic.rug.ac.be%3E

[29] "Apache Mailing List directory-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/directory-dev/201008.mbox/%3C4C6022F4.9020806@gmail.com%3E

[30] "GitHub Issue of revapi/revapi." [Online]. Available: https://github.com/revapi/revapi/issues/81/

[31] "GitHub PR of habitat-sh/habitat." [Online]. Available: https://github.com/habitat-sh/habitat/pull/3374

[32] "GitHub Issue of riot-os/riot." [Online]. Available: https://github.com/RIOT-OS/RIOT/issues/2838/

[33] "StackOverflow discussion 30596247." [Online]. Available: https://stackoverflow.com/questions/30596247/maven-docs-after-codehaus-terminated

[34] "GitHub Issue of realm/jazzy." [Online]. Available: https://github.com/realm/jazzy/issues/670/

[35] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200007.mbox/%3C20000724115642.66550.qmail@locus.apache.org%3E

[36] "GitHub PR of gwpy/gwpy." [Online]. Available: https://github.com/gwpy/gwpy/pull/747

[37] "Apache Mailing List stdcxx-user." [Online]. Available: http://mail-archives.apache.org/mod_mbox/stdcxx-user/200901.mbox/%3C21587519.post@talk.nabble.com%3E

[38] "Apache Mailing List hc-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/hc-dev/200308.mbox/%3C1059775720.2453.91.camel@kczrh-okt22.corp.bearingpoint.com%3E

[39] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200208.mbox/%3C200208080345.g783jP217329@Boron.MeepZor.Com%3E

[40] "GitHub PR of rails/rails." [Online]. Available: https://github.com/rails/rails/pull/26288

[41] "GitHub PR of paulcollett/vue-masonry-css." [Online]. Available: https://github.com/paulcollett/vue-masonry-css/pull/7

[42] "GitHub PR of composewell/streamly." [Online]. Available: https://github.com/composewell/streamly/pull/37

[43] "GitHub Issue of azure/azure-sdk-for-php-samples." [Online]. Available: https://github.com/Azure/azure-sdk-for-php-samples/issues/8/

[44] "GitHub PR of facebook/react-native." [Online]. Available: https://github.com/facebook/react-native/pull/8566

[45] "StackOverflow discussion 31372056." [Online]. Available: https://stackoverflow.com/questions/31372056/doxygen-unable-to-load-isutils-dll

[46] "GitHub Issue of commercialhaskell/stack." [Online]. Available: https://github.com/commercialhaskell/stack/issues/2620/

[47] "StackOverflow discussion 532779." [Online]. Available: https://stackoverflow.com/questions/532779/is-there-anything-like-ghostdoc-for-c

[48] "StackOverflow discussion 1136234." [Online]. Available: https://stackoverflow.com/questions/1136234/is-there-a-ruby-documentation-tool-that-allows-inclusion-of-diagrams-and-images

[49] "StackOverflow discussion 48435375." [Online]. Available: https://stackoverflow.com/questions/48435375/how-to-know-the-possible-response-when-using-android-frameworks-method

[50] "GitHub PR of uber/luma.gl." [Online]. Available: https://github.com/visgl/luma.gl/pull/84

[51] "StackOverflow discussion 23900027." [Online]. Available: https://stackoverflow.com/questions/23900027/automodule-breaking-with-python-return-type-annotations

[52] "GitHub Issue of pinax/pinax-badges." [Online]. Available: https://github.com/pinax/pinax-badges/issues/2/

[53] "GitHub PR of netflix/hollow." [Online]. Available: https://github.com/Netflix/hollow/pull/63

[54] "Apache Mailing List camel-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/camel-dev/201701.mbox/%3Cgit-pr-1380-camel@git.apache.org%3E

[55] "StackOverflow discussion 45737685." [Online]. Available: https://stackoverflow.com/questions/45737685/how-to-get-back-to-previous-numbering-value-and-indentation-after-promoting-a-le

[56] "GitHub Issue of phpdocumentor/phpdocumentor2." [Online]. Available: https://github.com/phpDocumentor/phpDocumentor/issues/1679

[57] "StackOverflow discussion 23689297." [Online]. Available: https://stackoverflow.com/questions/23689297/javadoc-8-cant-find-class-in-see-reference-but-java-7-did

[58] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200103.mbox/%3CBF19ACC2C98EDF43B1825F3FB5DE6B841E39E4@cs05ae01.cs05.danfoss.net%3E

[59] "GitHub Issue of dvajs/dva." [Online]. Available: https://github.com/dvajs/dva/issues/1275/

[60] "Apache Mailing List tomee-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/tomee-dev/201304.mbox/%3CCACLE=7MgFxVJnqC_2qQ7tKe-xPA=aCYxizh6NpvehnWQEr8FNw@mail.gmail.com%3E

[61] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200405.mbox/%3Cs098c7e0.071@sinclair.provo.novell.com%3E

[62] "GitHub PR of keratin/authn." [Online]. Available: https://github.com/keratin/authn/pull/38

[63] "Apache Mailing List jclouds-user." [Online]. Available: http://mail-archives.apache.org/mod_mbox/jclouds-user/201308.mbox/%3C80D2B5DD-5F06-41E9-A50F-D7E737547685@rackspace.com%3E

[64] "Apache Mailing List tomee-users." [Online]. Available: http://mail-archives.apache.org/mod_mbox/tomee-users/201407.mbox/%3CCAGV8jqjJ1tEbu=Xs8AkH0xRLZy7ekv+W91nj7gkpTcuyh5NgRw@mail.gmail.com%3E

[65] "Apache Mailing List jena-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/jena-dev/201309.mbox/%3CCAOQrJk4d_tDo_MBpkQVg0kFx1ruY0_q5OStnk8bQu1KMvoJ8rw@mail.gmail.com%3E

[66] "StackOverflow discussion 45342178." [Online]. Available: https://stackoverflow.com/questions/45342178/jsdoc-how-to-document-array-of-arrays-and-boolean-or-other-types-for-returns

[67] "Apache Mailing List tuscany-user." [Online]. Available: http://mail-archives.apache.org/mod_mbox/tuscany-user/200609.mbox/%3C451D9F2A.3000306@qwest.net%3E

[68] "GitHub PR of benjaminkott/bootstrap_package." [Online]. Available: https://github.com/benjaminkott/bootstrap_package/pull/211

[69] "Apache Mailing List httpd-docs." [Online]. Available: http://mail-archives.apache.org/mod_mbox/httpd-docs/200404.mbox/%3C200404220345.i3M3jrh31566@Boron.MeepZor.Com%3E

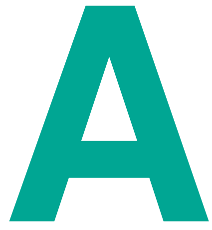
[70] "Apache Mailing List pig-user." [Online]. Available: http://mail-archives.apache.org/mod_mbox/pig-user/201209.mbox/%3C4819756062472907279@unknownmsgid%3E

[71] "Apache Mailing List systemml-dev." [Online]. Available: http://mail-archives.apache.org/mod_mbox/systemml-dev/201609.mbox/%3CCAGU5spdLvDDSxP9uhZO75BCMKgOJwVyL8eZaQZvJAHVJGGH4Qw@mail.gmail.com%3E

[72] "GitHub PR of prestodb/tempto." [Online]. Available: https://github.com/prestodb/tempto/pull/16

Part IV

Appendices



Code Time Machine

Exploring and analyzing the history of changes is an intrinsic part of software evolution comprehension. Existing tools that exploit the data residing in version control repositories provide only limited support for the intuitive navigation of code changes from a historical perspective. We present *Code Time Machine*, a lightweight IDE plugin which uses visualization techniques to depict the history of any chosen file augmented with information mined from the underlying versioning system. Inspired by Apple's Time Machine, our tool allows both developers and the system itself to seamlessly move through time.

Structure of the Appendix

- **Section A.1** provides the motivation for this work.
- **Section A.2** details our tool and explain its main components.
- **Section A.3** demonstrates our tool through a real-world scenario.
- **Section A.4** presents related work, and finally **Section A.5** concludes this appendix.

Supplementary Material

The source code of our tool, *Code Time Machine* plugin, as well as its binary release for IntelliJ IDE is publicly available (see below this page for download link). In addition, a demo of the tool can be found online[1].

Accomplishments in a Nutshell

The Code Time Machine [AMBL17]
Emad Aghajani, Andrea Mocci, Gabriele Bavota, Michele Lanza
In *Proceedings of 25th IEEE International Conference on Program Comprehension (ICPC 2017)*, pp. 356–359. IEEE, 2017.

***Code Time Machine*IntelliJ plugin**
Emad Aghajani, Andrea Mocci, Gabriele Bavota, Michele Lanza
publicly available for download at `https://github.com/emadpres/CodeTimeMachine`

[1] See `https://youtu.be/cBctQbjlAFY`

A.1 Introduction

Software evolution comprehension is an integral part of the software development process. According to Gîrba and Ducasse [GD06], a class of techniques to support software evolution analysis is *version-centered*. Such approaches target answering questions of when something happened in the history of a software project, and involve activities such as comparing two different versions, in terms of source code and/or runtime behavior. To perform such analyses, developers must be able to easily revert the system back to any given revision, perform static code inspections (*i.e.,* visit the source code and its corresponding metadata), compile and run the project, and possibly inspect the behavior at runtime.

In practice, the common facilities exposed by version control systems are rather limited. In fact, carrying out version-centered analyses is cumbersome, since it must be done through repetitive procedures using default user interfaces, such as the command line or applications with rather simple GUIs (*e.g., GitHub Desktop*[2]).

Besides source code history, additional information such as code metrics are the most important resources to understand a system's evolution. As opposed to reverse engineering where one needs to understand the current version of a system, understanding a system's evolution copes with such data multiplied by the number of its revisions [LDS01]. Although code metrics and the versioning system's data are accessible separately, *i.e.,* through different tools, the user himself has to manually collect and correlate them to perform a given evolutionary analysis.

According to Gonzalez-Torres *et al.* [GTGPT11], understanding the evolution of a software project can be effectively performed with the support of a visual representation. Several approaches have been proposed to understand a system evolution with the help of software visualization, for example by combining it with software metrics [LD02].

Consequently, different tools have been developed, primarily focusing on visualizing history to support the analysis of system evolution. *Gource* [Cau10], *CodeSwarm* [OM09], *SVN time-lapse View*[3] (and its Git version[4]), CVSScan [VTVW05], and *GitX*[5] are some examples of them. Such tools provide a common core of features, like a slide-bar for viewing different revisions of a file and a diff-view of changes. Although these tools could potentially be used to analyze a system's evolution, they do not uniformly integrate complementary information like code metrics into their visualizations.

More importantly, these tools do not provide any intuitive mechanism for previewing different versions of a system and navigate through them. The current practice is to perform the *checkout* operation for reverting to a specific version, which in the case of uncommitted changes would be problematic. In addition, in some scenarios, the developer analyzing a project's evolution needs to focus on a certain file in the course of time, but this is not possible because some of the tools do not provide a file-centric history view. Thus, to follow the evolution of a particular file, developers must manually find the commits which include the file.

To overcome the aforementioned problems, inspired by Apple's Time Machine, we came up with the idea of supporting the analysis of a system's evolution with a visualization that leverages the screen depth as the time axis. This concept brings a uniform, version-centered, and seamless history exploration experience to developers. Our visualization is file-centered, and it augments the familiar code editor of the IDE by enabling developers to navigate a file's history along a depth-based, perspective timeline, without losing the current context. In addition, the visualization integrates the following features:

[2] See `https://desktop.github.com/`
[3] See `https://code.google.com/archive/p/svn-time-lapse-view/`
[4] See `https://github.com/JonathanAquino/git-time-lapse-view`
[5] See `http://gitx.frim.nl/`

- a *code metrics view* to illustrate the evolution of metrics like Lines of Code (LOC), number of methods, and cyclomatic complexity;
- a zoomable *timeline view* that represents the history of commits for the file, and where the user can select an active range window;
- a detailed *commit list view* that enables inspection and navigation through the commits in the selected active range.

The visualization is implemented as the *Code Time Machine*, a lightweight language-independent plug-in for the IntelliJ Idea IDE.

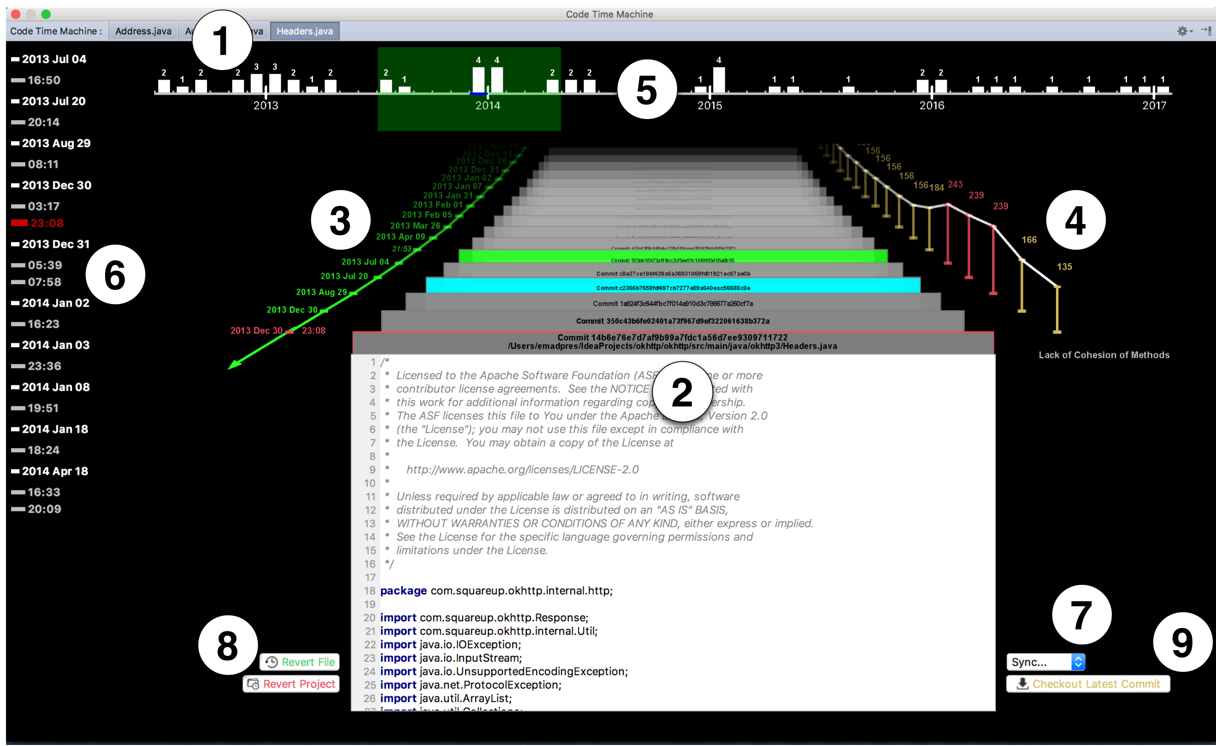


Figure A.1. Code Time Machine main window.

A.2 The Code Time Machine in a Nutshell

Figure A.1 depicts the main window of the *Code Time Machine*. A tab list represents the list of running *Code Time Machine* instances on different files (Figure A.1-①). The *commits stack view* (Figure A.1-②) depicts the history of underlying commits for the file and enables developers to simultaneously explore the evolution of source code and corollary code metrics. In case the file has uncommitted changes, a *virtual commit* is created for the sake of consistency. Each commit window in the stack represents a single version of the file. By hovering on a commit window, details about that commit are displayed on the top of the window (Figure A.2).

The *perspective timeline view* displays the commit's time (Figure A.1-③) and the *metrics view* shows the evolution of values of code metrics (Figure A.1-④). The developer can pick 14 metrics such as Lines of Code (LOC), number of methods, and cyclomatic complexity. The metric values are computed on the fly with no need for any preprocessing.

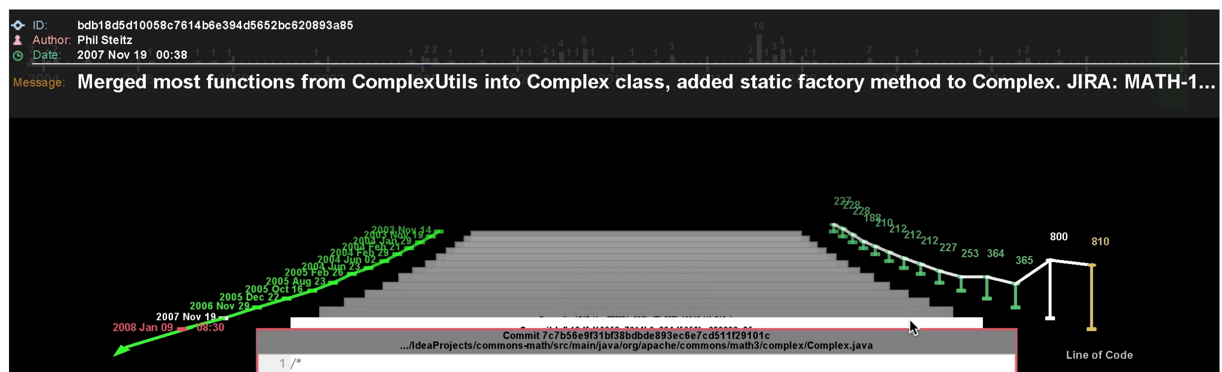


Figure A.2. Displaying commit information by hovering over one

The *timeline view* (Figure A.1-⑤) can be used to explore the commits at different levels of granularity. At the highest level (Figure A.3-Ⓐ), a vertical bar represents the total number of commits for each month. By zooming in, the timeline gets more detailed and commits are displayed individually (Figure A.3-Ⓑ).

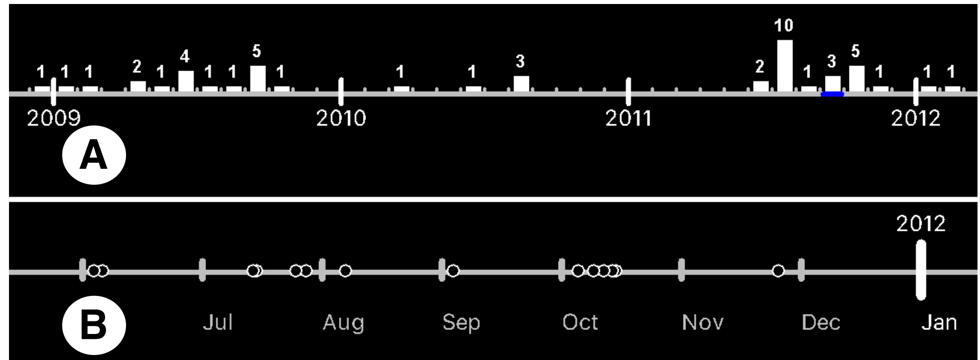


Figure A.3. Different levels of detail in *timeline view*

The *commit list view* (Figure A.1-⑥) along the *timeline view* provides an alternative mechanism to explore commits. A developer may select a specific period of time, *i.e.,* an *active range* displayed in green, by clicking and dragging on *timeline view*. Thus, the *commit list view* gets updated to show only the commits pertaining to the active range (Figure A.4).

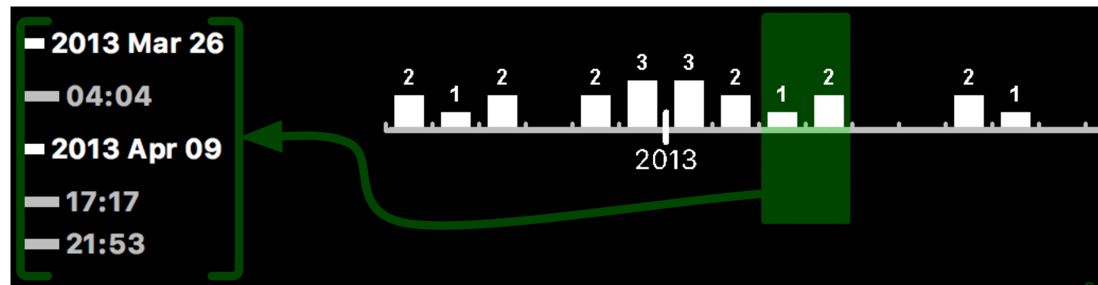


Figure A.4. Updating the *commit list view* by specifying active range

A developer can hover over any commit in the *commit list view* to inspect the commits' details on top of the window.

A.2.1 History Exploration

Our tool supports many ways to explore commits. First, a developer can use *timeline view* and *commit list view* to focus on a specific range of commits. In addition, she may start flying over all commits looking for interesting value changes in any of the available code metrics. After finding a range of interest, she can start navigating through commits one by one either using the keyboard or the mouse wheel.

Diff view. In the commit stack view, commits can be marked using keyboard keys ‘B’ and ‘N’: Figure A.1-② shows an example of marked commits, which are colored in cyan and green. After marking any two commits and pressing the space bar, a fully featured *Diff window* is displayed in a separated window, without losing the current context (see Figure A.5).

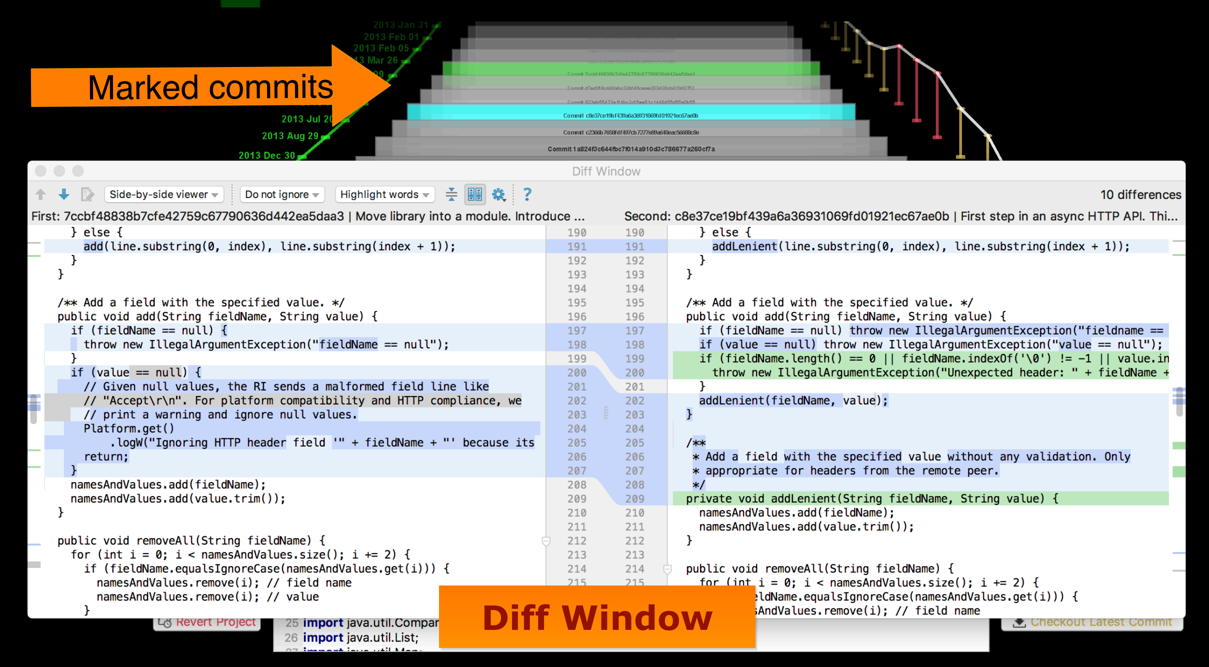


Figure A.5. The *Diff window*

Distinguishing Commit Authors. This feature allows developers to distinguish the commits made by a specific author. After enabling it, the header of the commits stack windows will be displayed in the same color when it belongs to the same author (see Figure A.6).

Commits Stack view Customization. Using keyboard keys ‘I’/‘K’ and ‘O’/‘L’, a developer can adjust the perspective of the *commits stack view* in terms of the maximum visible depth and the distance between commits windows, respectively. Figure A.6 also shows two different configurations of the perspective.

Commit Sync. The sync drop-down list (Figure A.1-⑦) can be used to synchronize time between two *Code Time Machine* instances representing two files. By pressing the “Sync” button and choosing one of the other available instances, the current view flies to the same commit (or the nearest one in the past if it does not exist) where the other instance is focused on.

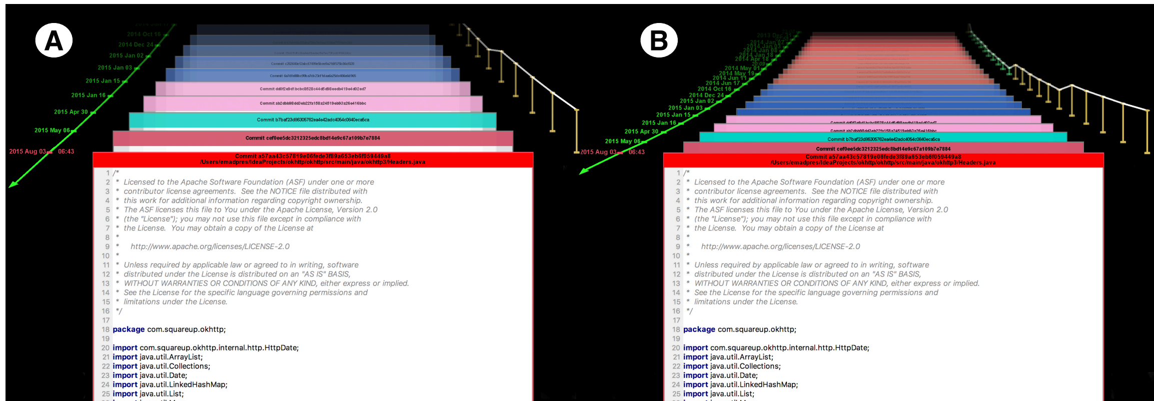


Figure A.6. Example of *commits stack view* customizability and colorful mode

A.2.2 Time Traveling

Figure A.1-⑧ shows the UI components dedicated to time traveling. Pressing the *Revert File* button reverts the underlying file to the currently selected commit, *i.e.,* the topmost window's source code. The *Revert Project* button reverts the whole project to the selected commit.

Both functionalities keep the developer's uncommitted changes safe. After clicking either buttons, the *Code Time Machine* window will get closed to let developers compile and run the project, for example to inspect the runtime behavior of the application in the selected revision.

To revert the whole project back to the latest commit, a developer can click the *Checkout Latest Commit* button (Figure A.1-⑨). Any previously uncommitted change is still available in the commit list, and a developer may restore it for a single file or for the whole project by selecting the corresponding virtual commit and using the revert buttons (Figure A.1-⑧).

A.3 The Code Time Machine In Action

Consider a scenario where a developer, Emma, wants to analyze the evolution of a class within the *Apache Commons Math*[6] project. As part of a reviewing activity, she focuses on the *Complex.java* file, that contains the class with the same name modeling a complex number.

By using the *Code Time Machine* she analyzes the evolution of specific code metrics in the history of the file: Starting from the very beginning of the file evolution, she spots a dramatic increase in the LOC, as shown in Figure A.2. Comparing the source code using the *Diff window*, she notices that the change relates to a major refactoring, moving methods from the *ComplexUtils* class to the *Complex* class.

To understand more, Emma decides to open the related files, the *ComplexUtils.java* file which is merged with the subject class, and the corresponding test classes *ComplexTest.java* and *ComplexUtilsTest.java*. She moves to the same commit time corresponding to the refactoring in the subject class using the "Sync" button in all of them.

Comparing the *ComplexUtils.java* at that moment, she finds out that a large number of methods have been deprecated and are refactored to call back the *Complex* methods, as she expected. Besides, she finds out that the moved methods' unit tests have been added to *ComplexTest* class correspondingly.

[6]See `https://github.com/apache/commons-math`

However, she notices that the *ComplexUtilsTest.java* file was not modified, though the unit tests there could be safely deleted as *ComplexUtils* methods are just recalling the *Complex*'s methods and *ComplexTest* is now covering them.

To find out whether this class is refactored later or the developers forgot, she decides to move through *ComplexUtilsTest.java* LOC evolution. She spots the moment when the *ComplexUtilsTest.java* LOC has a dramatic drop. Comparing the source code, she confirms that the unit tests are deleted at this moment, 5 months after the fusion.

At this moment, Emma is wondering whether there was some logic behind the decision of keeping *ComplexUtilsTest* unit tests or not. She speculates that the developers forgot to delete them. To support her hypothesis, she decides to run the pruned *ComplexUtilsTest* on the *ComplexUtils* class of the early refactoring time. Thus, first, she reverts the project to the commit time corresponding to the fusion time. She runs the tests and they all pass, as it is expected. Then, she reverts the *ComplexUtilsTest.java* file to the 5-months-newer commit when unit tests are removed. By running the tests again, she observes that the tests still pass, which it means most probably the developers could have refactored the *ComplexUtilsTest.java* right after fusion.

A.4 Related Work

There are a number of studies that leverage visualization techniques to understand system evolution. The *Revision Towers* approach [TM02] models the evolution of a file telling by whom and to what extent a file has been changed. The *RepoGrams* [RBK+16] provides a metric-based visualization model to understand the metrics evolution during the history of a software project. Khan *et al.* [KBEL12] surveyed the research related to software architecture visualization. Ogawa and Ma [OM10] use a heuristic approach to present repository evolution focusing on the developers which are involved.

Some researchers have used the matrix as the baseline for their visualizations, such as a two-dimensional matrix to present metrics changes and to improve the software evolution comprehension [Lan01, LD02, LDGP05]. Tymchuk *et al.* [TMGN16] propose a visualization to detect quality fluctuations using a three dimensional matrix.

A popular technique has been the leveraging of a city metaphor to depict software systems in 3D [WL08]. There are also a number of other works that use the time concept to enable one to move back and forth through changes [HLL10, HDLL11].

To the best of our knowledge, our tool is the first to bring the code and code metrics seamlessly next to each other for system evolution understanding.

A.5 Conclusion and Future Work

The Code Time Machine aims to help system evolution comprehension with the support of visualization, fully integrated in the IntelliJ IDE. It provides a uniform code and code metrics history exploration and enables developers to revert a file or a system to a specific version seamlessly.

The main view of the tool integrates a variety of different code metrics right next to the source code. The tool also enables developers to mark two versions and compare their source code to figure out how or why the value of a code metric has changed at some point in time.

As part of our future work, we plan to extend the file history view and enable developers to have the same seamless history navigation experience in terms of package and project scope.