How Do Developers Discuss Rationale?

Rana Alkadhi*, Manuel Nonnenmacher*, Emitza Guzman†, and Bernd Bruegge*

*Technical University of Munich
Department of Informatics
Garching b. München, Germany
{alkadhi, nonnenma, bruegge}@in.tum.de

†University of Zurich
Department of Informatics
Zurich, Switzerland
guzman@ifi.uzh.ch

***Abstract*—Developers make various decisions during software development. The rationale behind these decisions is of great importance during software evolution of long living software systems. However, current practices for documenting rationale often fall short and rationale remains hidden in the heads of developers or embedded in development artifacts. Further challenges are faced for capturing rationale in OSS projects; in which developers are geographically distributed and rely mostly on written communication channels to support and coordinate their activities. In this paper, we present an empirical study to understand how OSS developers discuss rationale in IRC channels and explore the possibility of automatic extraction of rationale elements by analyzing IRC messages of development teams. To achieve this, we manually analyzed 7,500 messages of three large OSS projects and identified all fine-grained elements of rationale. We evaluated various machine learning algorithms for automatically detecting and classifying rationale in IRC messages. Our results show that 1) rationale is discussed on average in 25% of IRC messages, 2) code committers contributed on average 54% of the discussed rationale, and 3) machine learning algorithms can detect rationale with 0.76 precision and 0.79 recall, and classify messages into finer-grained rationale elements with an average of 0.45 precision and 0.43 recall.**

I. Introduction

Developers make various decisions during software development. The rationale behind these decisions including raised issues, proposed alternative solutions and arguments for or against implementing specific alternatives is of great importance during software maintenance and evolution. Rationale provides a profound reasoning of *why* the system is designed the way it is [26]. Documented rationale is a type of developer documentation, an umbrella term recently coined by Robillard et al. [36] for documents intended to assist developers in the creation and maintenance of the system. It is considered among the most useful information for developers during software maintenance [25]. Rationale helps developers understand the intent behind past decisions [20]. Time and development efforts can be saved by documenting alternatives visited and rejected earlier [13]. In addition, captured rationale enhances software artifacts traceability, change impact analysis and facilitates design verification, and knowledge sharing [21].

Missing rationale information has negative impacts on software development, as *"developers often had to defer tasks because the only source of knowledge was unavailable coworkers."* [20]. However, developers are often hesitant to capture rationale explicitly due to time and cost constraints and rationale remains tacit in developers' heads or embedded in development artifacts [13]. Capturing rationale becomes more challenging in Open Source Software (OSS) in which developers are geographically distributed and no clear design documentation can be detected [6]. Developers in OSS projects rely heavily on written communication to collaborate and coordinate their development activities such as IRC channels, mailing lists, and Wikis. Developers' communications occurring over these channels contain a wealth of information about the software system such as design decisions [18], [20], development history, and rationale [1], [2], [44].

Internet Relay Chat (IRC) channels are increasing in popularity for synchronous communications in OSS projects [7], [11], [19], [43]. Developers use IRC channels for discussing development and implementation details and exchanging knowledge and ideas with other developers [17], [45]. Mozilla Foundation describes IRC as *"the primary form of communication for members of the Mozilla community"*[1].

While several prior studies have examined the general role of IRC channels in OSS development [7], [17], [43], there is still no empirical evidence of how OSS developers discuss rationale in IRC messages. The following excerpt from the Apache Lucene website reinforces our motivation for studying rationale in developers' IRC messages.

> *"The IRC channel can be used for online discussion about Lucene related stuff, but developers should be careful to transfer all the official decisions or useful discussions to the issue tracking system."*[2]

This excerpt sheds light on two important aspects. First, developers' discussions over IRC channels might contain valuable rationale about development decisions. Second, there is no systematic methodology for transferring such knowledge to official documentation artifacts (e.g., issue trackers for most OSS projects) other than relying on the developers to transfer it manually. As a consequence, potential rationale lying hidden in IRC messages is rarely made explicit or taken advantage of.

In this paper, we present the results of an empirical study conducted on IRC chat logs of three large open source projects. Our objective is to investigate how developers discuss rationale while communicating on IRC channels to analyze the potential

[1]https://developer.mozilla.org/docs/Mozilla/QA/Getting_Started_with_IRC
[2]https://lucene.apache.org/core/discussion.html

TABLE I: Rationale elements.

Rationale element (based on Kunz and Rittel taxonomy [22])	**Definition**
Issue	Problem that needs discussion and negotiation to be solved. An issue typically can not be resolved algorithmically and does not have a single correct solution.
Alternative	Possible solution that could address the issue under consideration.
Pro-argument	Positive reason supporting an alternative.
Con-argument	Negative reason against an alternative.
Decision	The alternative selected to resolve an open issue.

of automatically detecting rationale in these messages. The contribution of this paper is threefold. First, we provide quantitative evidence of rationale existence in IRC messages and the frequency of different rationale elements by manually analyzing 7,500 IRC messages from three OSS projects. Second, we explore which developers contribute to rationale in IRC messages. Third, we investigate the performance of different machine learning algorithms when automatically detecting rationale in IRC messages and classifying them into finer-grained rationale elements: issues, alternatives, arguments, and decisions. This study is a first step towards understanding how OSS developers discuss rationale in IRC messages.

II. Background and Related Work

In this section, we provide a brief background about rationale in software engineering and discuss related work on two areas: automated extraction of rationale and studies on IRC in open source projects.

A. Rationale in Software Engineering

Rationale is the justifications behind decisions [13]. Many representation models have been proposed in the literature to capture different concepts of rationale. Kunz and Rittle were the first to capture rationale as an issue model and proposed the well-known IBIS (Issue Based Information System) model [22]. Other representation models emerged over the years, for example, QOC (Question, Option and Criteria) [27], PHI (Procedural Hierarchy of Issues) [28], and DRL (Decision Representation Language) [24]. In this paper, we identify rationale elements based on Kunz and Rittle taxonomy [22]. We analyze the following rationale elements: issues, alternatives, pro-arguments, con-arguments, and decisions. The rationale elements and their definitions are listed in Table I. We focus on these rationale elements as they form the basis for many other rationale models.

B. Automated Extraction of Rationale

Capturing rationale is a challenging area in the field of rationale management. Manual methods for capturing rationale usually fail in practice due to high overhead. Another possible reason is the gap between the developers documenting rationale and the ones using it at later stages. This problem is commonly known in literature as the *capture problem* [13].

In a recent work, Robillard et al. [36] advocate for a new vision of an on-demand developer documentation (OD3), which promotes using development artifacts for the automated generation of developer documentation. The authors discuss its opportunities and challenges and view it as a promising research direction for software documentation. Sharing the same vision, several researchers have addressed exploiting available development artifacts for the automated extraction of rationale. Liang et al. [26] propose an algorithm for capturing design rationale from patent documents into a three layers model consisting of issues, design solutions, and artifacts layers. Myers et al. [30] propose a framework for producing a rich design history by recording designers' interactions. Rogers et al. [38], [40], [39] investigate the use of ontology and linguistic features for training machine learning models to classify rationale into decisions, alternatives, and argumentation from two types of documents: bug reports and design sessions transcripts. Pascarella and Bacchelli [35] propose a taxonomy for classifying code comments among which is implementation rationale. The authors apply machine learning algorithms for the automatic classification of code comments into the proposed taxonomy. Bhat et al. [3] propose a supervised machine learning approach for detecting and classifying architectural design decisions in issue tracking systems. Our work differs in that we focus on extracting rationale from short, unstructured developers' messages in IRC channels.

Brunet et al. [6] apply supervised machine learning techniques for the automatic identification of structural design discussions in issues, commits, and pull requests. In our work, however, we focus on analyzing more general knowledge about development rationale. A different perspective on studying rationale is presented in the work of Kurtanović and Maalej [23] in which the authors study rationale concepts in user online reviews and investigate various machine learning approaches for the automatic mining of user rationale. However, we focus on studying rationale from the developers perspective. In our previous work [2], we explore rationale in developers chat messages and the potential of using supervised machine learning techniques for the automatic extraction of rationale. The study presented in this paper differs in that we analyze IRC messages of open source communities, while in our previous work we studied chat messages exchanged during university projects in settings more similar to commercial, closed-source development.

C. IRC in Open Source Projects

Communication data in OSS projects is mostly recorded and made publicly available, which attracted researches to study their role in software development as a valuable source of knowledge [18]. Yu et al. [48] investigate the use of IRC (synchronous) and mailing list (asynchronous) communication mechanisms in global software development projects. They observe that developers actively use both as complementary communication mechanisms. A growing body of literature has analyzed developers communication over IRC channels. Shihab et al. [42], [43] investigate the IRC meetings content, participants, their contribution and communication styles. Elliott and Scacchi [15] show that open source communities

use IRC channels to mitigate and resolve conflicts and to build a community. Elliott [14] studies how cultural beliefs and values affect development processes in the organization. Chowdhury and Hindle [9] propose an approach for the automatic filtering of off-topic IRC discussions by exploiting StackOverflow programming discussions and YouTube video comments. Panichella et al. [34] investigate collaboration links by analyzing communication data from mailing lists, issue trackers, and IRC chat logs of seven OSS projects. Our work differs in that we analyze a specific type of knowledge, namely rationale, and the relation between development activities and rationale contribution in IRC messages.

III. Study Design

This section introduces the design of our empirical study. We introduce our research questions and describe the different phases of the applied research method: data collection, multiple alias resolution and the manual annotation process.

A. Research Questions

In our analysis, we aim to evaluate IRC messages as a source of rationale about the software system and to investigate the potentials of automated techniques to support developers in recovering rationale from these messages. In particular, we answer three research questions:

RQ1. Rationale frequency: **How often do OSS developers discuss rationale in IRC messages?** While it is widely accepted that IRC messages in OSS projects contain valuable information about the software system and its history, there is still a lack of empirical evidence about the presence and volume of rationale in these messages. Answering this question provides insights about the nature of existing rationale and provides the basis for training and evaluating automated classification techniques on the annotated messages.

RQ2. Rationale contributors: **Which developers contribute rationale in IRC messages?** This question is inspired by the work of Brunet et al. [6] that found a strong correlation between development activities (committing into code repository) and contributing to design discussions in pull requests, commits, and issues. By answering this question, we aim to find if such correlation holds between development activities and rationale contribution in IRC discussions; Do developers who commit more often contribute more rationale? This could provide first insights on linking the rationale found in IRC messages to different parts of the source code.

RQ3. Automatic classification: **How accurately can we classify IRC messages containing rationale by applying supervised machine learning algorithms?** In this question, we aim to evaluate different supervised machine learning techniques to detect and classify rationale into issues, alternatives, arguments, and decisions. Automated techniques will help developers exploit rational embedded in their IRC discussions, where manual analysis is often not feasible.

B. Research Method

Our research method consists of two phases: data collection and data analysis, as depicted in Figure 1. In the data collection phase, we first select the OSS projects for our study. We selected three OSS projects: Apache Lucene, Mozilla Thunderbird, and Ubuntu. In accordance with selection criteria applied by similar studies [33], [34], [43], we chose the three OSS projects for the following reasons. First, to mitigate threats to external validity, we selected projects from diverse domains. Second, the archived IRC logs and source code repositories of the selected OSS projects are publicly available. Third, the three projects are popular and mature OSS projects with a large community of active developers and users. After selecting the projects, we crawled the IRC chat logs and source code repositories of the three projects to extract the IRC messages and project committers. We detail on each step in Subsection III-C.

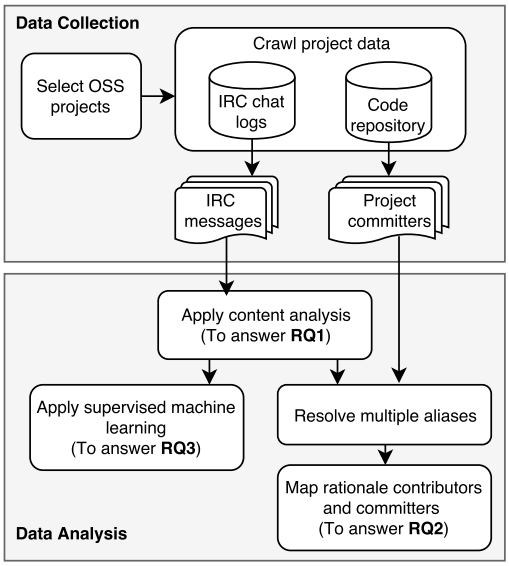


Fig. 1: Applied research method.

To answer ***RQ1***, we first annotate a sample of 7,500 IRC messages by applying content analysis techniques as described by Neuendorf [32]. The message annotation process is explained in Subsection III-E. To answer ***RQ2***, we map IRC authors who contributed rationale with the project committers. This process consists of two steps. First, we apply an alias resolution approach on IRC and committers identifiers separately, as developers can use multiple identifiers within a single channel. Second, we associate the IRC identifiers with the identifiers used in the commit history whenever viable. Both steps are elaborated further in Subsection III-D. Finally, to answer ***RQ3***, we use the annotated IRC messages as a training and validation set to build supervised machine learning classifiers. We compare the performance of different learning algorithms and classification configurations in Section VI.

C. Data Collection

In this section, we describe the process of collecting and extracting our research data from three OSS projects: Apache Lucene, Mozilla Thunderbird, and Ubuntu. For each project, we crawled the IRC logs and commit history, parsed the data to extract the required fields, and stored them into a

TABLE II: Overview of the research data.

Project	IRC messages				Commit history		Annotation sample (#IRC messages)
	Years	Channel	#Messages (before filtering)	#Messages (after filtering)	Years	#Commits	
Apache Lucene	2010-May 2017	*#lucene-dev*	273,123	71,897	2001-2017	27,787	2,500
Mozilla Thunderbird	2012-May 2017	*#maildev*	299,771	291,416	2007-2017	20,569	2,500
Ubuntu	2004-May 2017	*#ubuntu-devel*	2,897,987	2,438,812	1999-2017	349,813	2,500
Total	—	—	3,470,881	2,802,125	—	398,169	7,500

MySQL database for further analysis. For each IRC message, we extracted: `message author`, `message`, and `date`. For each commit, we stored: `committer`, `commit message`, and `commit date`. An overview of our research data is shown in Table II. In total, we collected 3,470,881 IRC messages and 398,169 commits from the three projects. We detail on the collection process for each project in the following.

Apache Lucene is a Java-based full-text search engine library. In recent years, it has become one of the most popular free information retrieval libraries [29]. We obtained the complete archive of the development IRC channel *#lucene-dev* logged by Colabti[3]. Next, we filtered out automatically generated messages, for example, messages generated when users join or leave the channel (*"*** hoss joined"*). This resulted in 71,897 messages written by 266 authors over the last 8 years. From Lucene's code repository, we collected 27,787 commits done by 152 committers over the last 16 years. We also obtained the official list of committers from Lucene's website.

Mozilla Thunderbird is a cross platform email client with an estimation of 25 million active users[4]. We crawled 299,771 IRC messages from the development channel *#maildev* logs. Afterwards, we filtered out messages posted by Firebot (a general-purpose Mozilla chatbot) which resulted in 291,416 messages written by 1,180 authors over the last 6 years. From Thunderbird's code repository, we collected 20,569 commits performed by 895 committers over the last 11 years. Additionally, we imported the official list of core developers provided by Thunderbird.

Ubuntu is a Debian-based Linux operating system. Ubuntu is one of the most popular Linux distributions and has over 40 million desktop users and more than 500 active members from 100 countries. The development IRC channel *#ubuntu-devel* is *"home to many Ubuntu developers for real-time communication"*[5]. We fetched the complete channel archive[6] and filtered out automatically generated messages such as *"=== bob2 [rob@bob2.user] has joined #ubuntu-devel"*. This resulted in 2,438,812 messages written by 13,645 authors over the last 14 years. Likewise, we extracted 349,813 commits done by 6,724 committers over the last 18 years. Finally, we queried the official list of core developers provided by Ubuntu.

[3]http://colabti.org/irclogger/irclogger_logs/lucene-dev
[4]https://blog.mozilla.org/thunderbird/
[5]https://wiki.ubuntu.com/UbuntuDevelopment
[6]https://irclogs.ubuntu.com

D. Alias Resolution

> *"I wish we didn't have these pseudo-names here; I don't know who's who half the time."*
>
> — Apache Lucene Developer

The multiple alias problem occurs when multiple nicknames (aliases) are assigned to the same person [42]. It is commonly faced in studies examining OSS repositories [4], [16], [37], [43]. Resolving this problem is an essential step to prepare our data for further analysis. The cause of this problem in our study is twofold. First, we use multiple data sources in our study, namely IRC channels and commit history, and developers might use different identifiers on these sources. In IRC channels, participants assign themselves nicknames when joining the channel. A nickname is a self-chosen name [5], which can be an abbreviation of the real name (e.g., *markmiller* for *Mark Miller*), or a pseudonym (e.g., *luceneuser*). While in source code repositories, developers use names, nicknames, emails, or a combination of them. Second, developers might use multiple aliases within a single source; mostly very similar ones. For example, *sagarwal*, *sshagarwal* and *sshagarwaltb* are used by the same person to write to IRC. Similarly, a developer can commit code to a repository with different identifiers. For example:

Michael McCandless <mikemccand@apache.org>,
Michael McCandless <mail@mikemccandless.com>,
Mike McCandless <mikemccand@apache.org>, and
mikemccand <mike@elastic.co>.

To resolve aliasing in our collected data, we applied an approach similar to the ones by Bird et al. [4] and Panichella et al. [34]. We started with the extracted IRC `message authors` and `committers` from the three projects (Section III-C). For each project, we performed the following steps automatically:

1) **Extract email login names:** We removed emails' domains (anything after "@"). For example, *sarowe@gmail.com* and *sarowe@apache.org* are converted to *sarowe*.
2) **Normalization:** We converted identifiers to lowercase, removed punctuation (e.g., "_"), numbers and eliminated extra whitespace. For example, *JoeS*, *JoeS1* and *JoeS11* are all mapped to *joes*. Additionally, we removed the

term *"–guest"* that was commonly attached to Ubuntu IRC identifiers (e.g., *yeager-guest*).

3) **Ignore middle names:** We removed middle names and initials if real names were provided. For example, *Jory A. Pratt* is converted to *Jory Pratt*. We use *'real names'* to refer to the names that were used together with the emails or nicknames to commit code. There is no guarantee that they are the developers' actual names and they can be also abbreviated versions of their names.
4) **Name similarity:** We applied a string similarity algorithm, Levenshtein edit distance [31], [47], to resolve aliases within IRC and committers identifiers separately. We set a conservative similarity threshold of 80%.
5) **Email-like similarity:** If multiple slightly different names have the same email login names, we considered them a match. For example, *nicholas knize* and *nick knize* both have the email login name *nknize*. We excluded commonly used email login names such as *mozilla@email_domain*.
6) **IRC authors-committers mapping:** We mapped the final list of IRC identifiers to committers identifiers to detect IRC committers. For a more accurate mapping, we consolidated the available information with the additional identifiers provided on the official lists of contributors on the project website.

Manual corrections were applied for some cases. An overview of the IRC authors and committers for each project after resolving aliases is shown in Table III. Although, we were able to resolve the majority of aliases within the single source (i.e., IRC and commit history), the mapping of IRC authors to project committers was not feasible in some cases. The main reason is that only the nicknames of IRC authors were available, while in most cases names or emails were used for committing code.

TABLE III: IRC authors and Committers.

Project	#IRC authors	#Committers	#IRC committers
Apache Lucene	221	107	26
Mozilla Thunderbird	1,029	633	83
Ubuntu	10,657	5,961	186

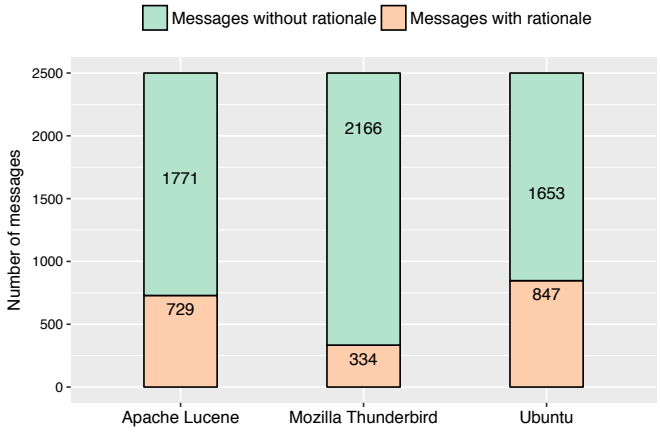


Fig. 2: Content analysis results.

E. IRC Messages Annotation

To analyze the frequency of the rationale elements in IRC messages, two authors of this paper applied manual content analysis [32] on a sample of IRC messages from the three studied projects. The manual annotation process consists of the following steps:

1) Annotation guide: To systematize the annotation process and assure a common understanding, we designed an annotation guide[7]. The annotation guide provides instructions about the annotation task and definitions and examples of the different rationale elements (listed in Table I). It was developed in two iterations. In each iteration, the two annotators used the guide to annotate a random sample of 300 messages. The disagreements were analyzed and the guide was refined accordingly.

2) IRC messages sampling: IRC messages are short in length (μ=54.31, M=42, SD=48.89 characters), written in informal language and context-dependent. Analyzing messages in isolation of the context in which they were exchanged might lead to imprecise results. To keep the conversation context, we created our sample by randomly selecting complete chat days instead of single messages. For each project, we randomly selected chat days so that the total adds up to 2,500 messages. Overall, our sample consisted of 35 chat days from Apache Lucene, 10 days from Mozilla Thunderbird, and 8 days from Ubuntu. This results in a sample of 7,500 messages from the three OSS projects.

3) Manual annotation: To avoid bias during the annotation, each message was annotated by the two annotators independently. For each message, the annotators indicated if the message contains rationale and specified the type of rationale element(s) included in the message. A message can be annotated with more than one rationale element. For example, *"Maybe for later version support, we should do it like in StandardTokenizer"* is annotated with alternative and pro-argument. We used GATE [10] for the manual annotation of the messages. During the annotation task, the complete chat days were displayed for the annotators. This allowed the annotators to obtain the conversation context while annotating single messages. We refer to the annotation guide for the detailed annotation instructions[7]. Annotators reported an average of 20 hours to complete the annotation task.

4) Disagreements reconciliation: All message were annotated twice. We consider that disagreements occur when only one of the annotators annotated a message as containing rationale or when the two annotators annotated a message with different rationale elements. The average inter-rater agreement was 83% for identifying messages containing rationale and 78% for identifying different rationale elements. Annotators discussed and resolved their disagreements.

[7]https://cloudbruegge.in.tum.de/index.php/s/HREfUU0I3Ty8kEo

TABLE IV: Frequency of rationale elements per project.

Rationale element	Apache Lucene	Mozilla Thunderbird	Ubuntu	IRC example
Issues	27%	24%	36%	"[...] I think we should find a way to configure the real system packages properly; [...]"
Alternatives	37%	50%	33%	"I'd rather just see php7 in experimental ASAP, and then slowly work out all rdeps until they all seem to more or less work, then just transition [...]"
Pro-arguments	19%	25%	19%	"Maybe for later version support, we should do it like in StandardTokenizer"
Con-arguments	11%	18%	15%	"But if you have lots of data this could make your GC on Solr go wild."
Decisions	17%	4%	15%	"I had considered it, but it didn't fit with the design I had, so I ignored it [...]"

IV. Rationale Frequency

To answer ***RQ1*** on the rationale frequency in IRC messages, we report on the results of the manual content analysis (see Section III-E).

On average, 25% of the analyzed messages contain rationale with a total of 1,910 out of the 7,500 annotated messages. We found that rationale frequency varies among the analyzed OSS communities, as illustrated in Figure 2. Ubuntu messages contain the highest amount of rationale in our sample (34%), followed by Apache Lucene messages (29%) and Mozilla Thunderbird messages which contain the lowest amount of rationale (13%).

Messages of OSS projects contain over twice (25% on average) as much rationale as messages of closed-source projects (9% on average) [2]. A possible explanation for such increase is that OSS developers are geographically distributed and working across various time zones. Thus, OSS developers communicate and discuss development issues in IRC messages more often than co-located development teams, in which regular face-to-face meetings are more common.

Table IV presents the frequencies and examples of fine-grained rationale elements. Alternative is the most prevalent rationale element in developers discussions with a total of 37% among the three projects. A possible explanation is that developers use IRC channels to discuss proposed alternatives with other developers, in the form of *"So I have some ideas and I want to get your opinion"*, more often than other rationale elements. There are other complementary channels in OSS projects for reporting issues or documenting final decisions, such as issue tracking systems, which might affect their frequency in IRC messages. However, the argumentative discussions of possible alternatives happen mostly through written communication channels in the absence of regular face-to-face meetings. For example, messages like *"Hi guys..I want to patch the issue [Issue#]"* in which developers reference an already opened issue to discuss their solution alternatives were commonly encountered during the manual annotation.

The second more frequent rationale element is issue with a total of 30%, followed by pro-arguments (20%), and finally con-arguments and decisions with an equal frequency of 14%. Developers tend to provide pro-arguments supporting their proposed alternatives, e.g., *"to make it 100% correct it would need to be volatile"*, which might explain the higher frequency of pro-arguments. The two annotators agreed that identifying decisions was the most difficult among other rationale elements. We noticed that the decisions are usually not clearly stated in the messages even when a consensus is reached.

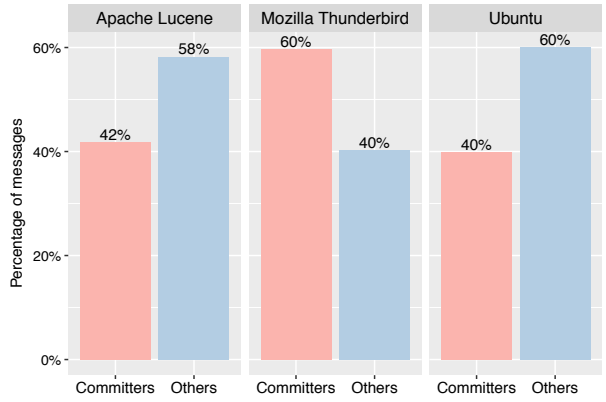
We found that 85% of the messages containing rationale only discuss one rationale element, 13% discuss two elements, and a few messages discuss three elements (1%). To gain further understanding on how developers discuss rationale in IRC messages, we analyzed the pair-wise co-occurrence correlation between different rationale elements at the message-level. In all three projects, there is a moderate negative co-occurrence correlation between issues and alternatives (Pearson's correlation <= -0.4), and with a lesser degree between issues and other rationale elements (Pearson's correlation <= -0.1). We also found a mild negative co-occurrence correlation between alternatives and decisions in all three projects (Pearson's correlation <= -0.2). Most developers communicate on IRC messages through informal short messages. The short length of these messages could explain the absence of any strong correlations between the different rationale elements, as messages when containing rationale most likely contain only one element.

V. Rationale Contributors

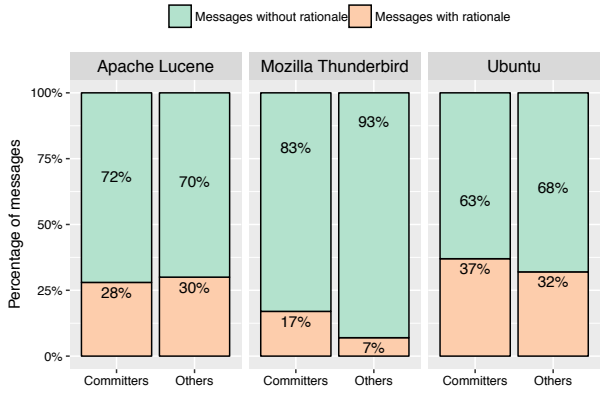
In this paper, we analyze messages from IRC channels dedicated to discussing development issues and we interpret the results with the underlying assumption that message authors are developers contributing to the OSS project. However, IRC channels are public and anyone can join the ongoing discussion. Answering ***RQ2*** provides the opportunity to investigate how participation in rationale discussions in IRC messages is related to committing actual code changes.

We distinguish between two types of IRC message authors (process described in Section III-D). *Committers* are developers who are committing to the project code repository and are identified from the project commit history. *Others* are the message authors who were not mapped into a committer name.

The analysis in this section is conducted on the analyzed message sample of 7,500 messages (2,500 messages from each project). From the IRC message authors, we identified 14 committers and 20 others for Apache Lucene, 27 committers and 43 others for Mozilla Thunderbird and 41 committers and 116 others for ubuntu. In all three projects, the number of other



(a) Messages contribution



(b) Rationale contribution

Fig. 3: Messages and rationale contribution of Committers and Others per project.

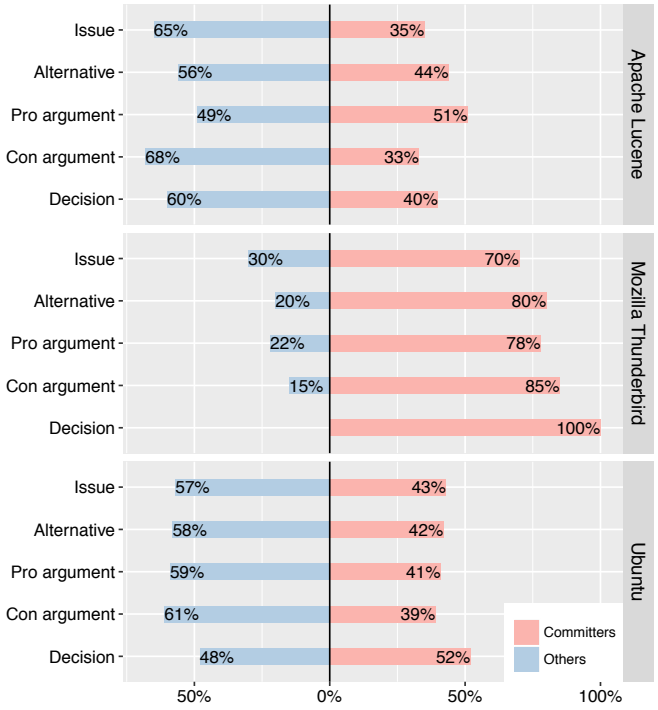


Fig. 4: Rationale elements distribution per project.

participants is larger than the number of committers. This is typically expected due to the public nature of IRC channels in OSS projects. However, the developers make efforts to keep the discussions in the development channels focused on development matters, e.g., *"[...] please ask support questions in #ubuntu; I realize that it's noisier there, but consider what it would be like here if everyone asked for user help here rather than in #ubuntu :-)"* is a developer reply to an end-user who posted a general question.

Figure 3(a) shows the number of messages written by each group of message authors. In both Apache Lucene and Ubuntu, the number of messages written by others exceeds the number of messages written by committers. While in Mozilla Thunderbird, committers wrote more messages than other authors even though the number of identified committers (27 committers) is less than other authors (43 others).

We found that the percentage of rationale contribution is proportional to the number of messages written by each group. In other words, the more messages written the more rationale contributed. Overall, committers contributed 40%, 78% and 44% of the rationale in Apache Lucene, Mozilla Thunderbird, and Ubuntu, respectively. On average, committers contributed 54% of the discussed rationale in the analyzed IRC messages.

Figure 3(b) displays the percentages of messages containing rationale among the messages contributed by each group. In both Mozilla Thunderbird and Ubuntu, the percentage of messages containing rationale by committers is higher than by others. In Apache Lucene, the percentages of messages containing rationale are similar for committers and others.

To deepen our understanding of how different groups contribute to rationale discussions, we analyzed the distribution of the rationale elements among committers and other authors, as shown in Figure 4. Committers contributed more decisions than other authors in both Mozilla Thunderbird and Ubuntu. A possible interpretation is that committers have more influence on the project development and authority to make decisions. However, messages written by other authors contain more decisions than the committers in Apache Lucene. For other rationale elements, there is no strong difference between the IRC author groups and the frequency of the rationale elements.

VI. Automatic Classification

In this section, we investigate the potential of supervised machine learning techniques for the automated extraction of rationale on two levels of granularity: binary and fine-grained classification.

Rationale binary classification focuses on the classification of messages into two categories: messages with rationale and without rationale. *Rationale fine-grained classification* focuses on the classification of messages with rationale into the fine-grained rationale elements: issues, alternatives, pro-arguments, con-arguments, and decisions. We performed the classification on the message level as previous work found it to be more

TABLE V: Binary classification results.

Classification algorithm	Messages with rationale			Messages without rationale		
	Precision	Recall	F1	Precision	Recall	F1
MNB	0.55	**0.69**	**0.61**	**0.88**	0.81	0.85
SVM	0.65	0.50	0.56	0.84	0.91	**0.87**
KNN	0.57	0.10	0.17	0.76	**0.98**	0.85
Decision Tree	0.50	0.32	0.39	0.79	0.89	0.84
Random Forests	**0.76**	0.14	0.23	0.77	**0.98**	0.86

TABLE VI: Binary classification results with different configurations: (1) Balanced dataset + 10-fold cross validation, and (2) Project cross validation (without balancing techniques).

Config.	IRC messages	MNB			SVM		
		Precision	Recall	F1	Precision	Recall	F1
(1)	With rationale	**0.76**	**0.79**	**0.77**	**0.76**	0.74	0.75
	Without rationale	**0.78**	0.75	**0.77**	0.75	**0.77**	0.76
(2)	With rationale	0.41	**0.46**	**0.39**	**0.54**	0.33	0.37
	Without rationale	**0.80**	0.77	0.77	0.79	**0.90**	**0.84**

accurate than sentence level classification when categorizing rationale elements from short developers' messages [2].

For training and validating the classifiers, we applied a 10-fold cross validation on the manually annotated IRC messages in Section III-E. For evaluating the performance of different machine learning classifiers, we use standard metrics in machine learning: precision, recall, and F-Measure (F1). They are calculated as follows: $Precision_i = \frac{TP_i}{TP_i+FP_i}$ and $Recall_i = \frac{TP_i}{TP_i+FN_i}$. Where TP_i is the number of messages that are correctly classified as being of type i, FP_i is the number of messages that are incorrectly classified as being of type i and FN_i is the number of messages that are incorrectly classified as not being of type i. The F-Measure is the harmonic mean of the precision and recall.

A. Rationale Binary Classifier

1) Setup: We preprocess the messages by converting the message text into lowercase, splitting it into single tokens and converting it into a vector space model by applying TF-IDF as a weighting method. We compare the performance of five classification algorithms: Multinominal Naive Bayes (MNB), Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Decision Tree and Random Forests. The experiments were performed using WEKA[8].

2) Data Balancing: A common challenge in supervised machine learning is imbalanced training datasets, which might cause the classification algorithm to skew towards the majority class and ignore the minority class. This problem is present in our dataset, in which only 25% of the messages contain rationale. We address this problem by applying a combination of over-sampling and under-sampling, common balancing techniques, as it proved to achieve better performance than applying each technique separately [8]. In particular, we apply SMOTE and random under-sampling on the annotated messages. SMOTE (Synthetic Minority Over-sampling Technique) [8] over-samples the minority class by generating synthetic examples and under-sampling [12] uses a subset of the majority class.

3) Project Cross Validation: For testing the generalizability of our classification models, we apply project cross validation. Project cross validation is a 3-fold cross validation with one fold per project. The classifier is trained on the messages of two projects and tested on the messages of the third project. The process is repeated three times rotating the projects and the averaged results are reported.

4) Results: Table V shows the classification results of the different classifiers. Overall, all the classifiers have a better performance when classifying messages without rationale. This result is expected due to the sparseness of the messages containing rationale in the annotated sample (25%). For the messages with rationale, MNB is the classifier with the best balance between precision (0.55) and recall (0.69) and the highest F-measure of 0.61. Examples of the classification results of the MNB binary classifier are shown in Table VII. The second best classifier is SVM with better precision than MNB (0.65) but lower recall (0.50). Random forests has the highest precision of 0.79, however, with a low recall of 0.14. Considering the small percentage of messages containing rationale, a classifier with a low recall is not desirable.

Table VI (1) shows the increase in the classification performance of messages with rationale when training the classifier on the balanced dataset, compared to the results in Table V. MNB has a better performance than SVM for the messages with rationale. However, the classification performance for messages without rationale decreased slightly as an expected result of under-sampling.

The averaged results of applying project cross validation are shown in Table VI (2) (without balancing techniques applied). The overall performance of the classifiers decreased comparing to the results when the messages from the same project are used for training the classifier. Nevertheless, these results show that the generated classifiers can be applied with a reasonable accuracy across projects.

B. Rationale Fine-Grained Classifier

1) Setup: We apply the same preprocessing steps in binary classification (Section VI-A1). Rationale fine-grained classification is a multi-label classification problem, in which a message can be classified into more than one rationale element (as explained in Section III-E). We compare between two popular transformation methods: Binary Relevance (BR) and Label Powerset (LP) [46]. We apply the two learning algorithms that have the better performance in the binary classification: MNB and SVM. The experiments were performed using MEKA[9].

2) Results: Table VIII provides an overview of the classification results of the different rationale elements. MNB has a better recall than SVM for all rationale elements when applying BR. On the other hand, SVM has a better precision in detecting all rationale elements. When comparing the F-measure, MNB

[8]http://www.cs.waikato.ac.nz/ml/weka

[9]http://meka.sourceforge.net

TABLE VII: Classification examples.

Classification	IRC message	Manual	Automatic
Binary	*"This bug was found because I reduced the maximum hit number randomly when collecting results"*	With rationale	With rationale
	"Never check email on waking up :)"	Without rationale	Without rationale
	"But if you have lots of data this could make your GC on Solr go wild"	With rationale	Without rationale
Fine-grained	*"So I've seen 2 modes of failure. The first is getting more than 1 doc back for query-by-id (and we always use update, so it should be impossible)"*	Issue	Issue
	"Also I find the Overseer loop here a bit hard to digest. For one thing, instead of the Boolean refreshClusterState I think it's clear enough to simply have clusterState be null as the check. Then it's clear what the job of that condition is to do."	Alternative, Pro-argument, Con-argument	Alternative, Pro-argument, Con-argument
	"This would be solved in part by doing the separate scanner class like current StandardTokenizer"	Alternative	Issue, Pro-argument

TABLE VIII: Fine-grained classification results.

Rationale element	Binary Relevance					
	MNB			SVM		
	Precision	Recall	F1	Precision	Recall	F1
Issue	0.45	**0.44**	**0.59**	**0.51**	0.40	0.44
Alternative	0.46	**0.60**	**0.52**	**0.54**	0.47	0.50
Pro-argument	0.30	**0.66**	0.41	**0.46**	0.38	**0.42**
Con-argument	0.20	**0.60**	**0.30**	**0.35**	0.23	0.28
Decision	0.19	**0.55**	**0.28**	**0.32**	0.23	0.27
	Label Powerset					
	MNB			**SVM**		
	Precision	Recall	F1	Precision	Recall	F1
Issue	**0.53**	0.38	0.44	0.48	**0.47**	**0.48**
Alternative	0.48	0.55	0.51	**0.54**	**0.57**	**0.56**
Pro-argument	**0.68**	**0.52**	**0.59**	0.44	0.37	0.40
Con-argument	0.25	**0.36**	**0.30**	**0.35**	0.22	0.27
Decision	**0.32**	**0.35**	**0.33**	0.29	0.25	0.27

performs better than SVM for all the elements except for pro-argument. When applying LP, no classifier achieved higher scores on all the accuracy measures. Classifying decisions and con-arguments has the lowest accuracy among different rationale elements. This is understandable considering the sparseness of these elements in the messages with rationale (only 14% as reported in Section IV).

Overall, MNB with LP has the best average F-measure of the rationale elements among the different classifiers. Table VII provides examples of correctly and incorrectly classified messages. The examples were produced by a BR and MNB fine-grained classifier. We observed that the classifier tends to assign more rationale elements than what is present in the message, provided that MNB has better recall than precision for all elements.

We explored other classification features such as Part-Of-Speech tagging, sentiment, and message length. These features did not have significant improvement on the classification performance and were not reported here for space limitation.

VII. Discussion

One of the main causes for the rationale capture problem is the additional overhead of writing it manually. Our long-term research goal is to minimize this overhead by developing automated techniques to support developers in capturing and linking rationale across different development artifacts. And eventually making the captured rationale available to use during different maintenance and evolution tasks. In this paper, we focus on developers' communication over IRC channels as one of the potential sources for extracting rationale. We discuss our findings by revisiting our research questions.

How often do OSS developers discuss rationale in IRC messages? On average, 25% of developers' IRC messages contain development rationale. Alternatives are the most discussed rationale elements, followed by issues, arguments and finally decisions. We also found that techniques analyzing rationale in IRC messages should consider the context of the exchanged conversation. During our analysis, we observed that developers discuss rationale in a sequence of short messages with mostly one rationale element. With this in mind, recovering rationale as chunks of conversations allows for a better comprehension of the argumentation flow leading to the decision.

The use of IRC messages as a complementary channel with other communication mediums is apparent. Consequently, the rationale is fragmented across these channels. References to emails, issue trackers, and commits were common cases. Exploiting these references can be a first step towards the linkage of the rationale across different channels. Messages discussing particular commits or code branches could be employed to extract traceability links between code fragments and the rationale related to them in IRC messages.

Which developers contribute rationale in IRC messages? A developer might discuss the rationale behind an implementation when communicating with other developers through IRC messages. Linking the development activities of developers to their discussion is a first step towards recovering the related rationale. To achieve this, it is necessary to identify OSS developers across multiple communication channels.

In this paper, we distinguished between two groups of IRC authors: Committers who are committing code to project repository, and *Others* whose IRC identifiers could not be mapped to committers names. Committers contributed on average 54% of the rationale in IRC messages. An interesting finding is that the volume of the rationale contributed by each group is correlated to the number of messages written by that group (i.e., *Committers* or *Others*) rather than *who* wrote the message. This posts an important question: *Who are the Others?* If they are contributing to the rationale discussions, could they not be developers themselves?

We applied a set of name resolution heuristics and the Levenshtein edit distance algorithm [31], [47] with a conservative similarity threshold of 80%. Within the single source, i.e., IRC channels and code repositories, we resolved an average of 20% aliases. However, on average, only 9% of IRC authors were mapped to committers. This result might be due to the fact that developers use different identities on different channels and linking these identities is not always feasible. For example, in some cases, very short versions of the developers' names are used in IRC channels complicating its mapping to an actual developer name, e.g., *mvg*. Also using pseudonyms is common, e.g., *lovemeblender* and *blackbug*. Future improvements to the resolution method could explore different values for the threshold and follow the automated approaches with a manual inspection for resolving ambiguous cases.

In the ideal case, identification methods such as the one proposed by Robles and Gonzalez-Barahona [37] should be applied to track and maintain awareness of developers' activities across different project repositories. Such methods should take developers' privacy into account while being designed and applied.

How accurately can we classify IRC messages containing rationale by applying supervised machine learning algorithms? The primary results of applying supervised machine learning techniques are promising for detecting rationale in IRC messages with 0.76 precision and 0.79 recall. For the fine-grained classification into different rationale elements, the classification performance varies according to the rationale element frequency in the analyzed message, i.e., the more instances of the rationale element in the training data the better the performance. Future work could investigate different techniques for improving the classification performance. For example, adding classification features on whether the neighbor messages contain rationale. We hypothesize that this will improve the classification accuracy as developers tend to discuss rationale in a sequence of consecutive messages. Another technique to explore is applying data augmentation approaches [41] to address the sparsity of rationale in IRC messages.

Threats to Validity. We based our definition of rationale elements on Kunz and Rittle's taxonomy in their well-known IBIS [22]. Although different rationale representations exist in the literature, the considered rationale elements are basic elements that are shared among most rationale models [13].

We analyzed the messages through manual analysis by human annotators which is a highly subjective process. To mitigate this threat, we applied a peer-annotation process in which each message is annotated by two annotators independently. Both annotators are graduate students with a software engineering background. Moreover, an annotation guide was developed and used during the annotation process. Another threat to validity is sampling bias. To mitigate this threat, we randomly selected a large sample of 7,500 messages from three OSS projects from three diverse domains.

We rely on an automated alias resolution approach to map IRC authors to committers. However, some aliases might remain unresolved. Also, the fact that developers can use freely-chosen nicknames might result in a missing mappings between IRC authors and committers. Regarding the generalizability of our results, we selected popular OSS projects with a large community of users and developers. We conducted project cross validation to test the generalizability of the generated classification models across different projects. We encourage further study replications on OSS projects of different sizes.

VIII. Conclusion

In this paper, we presented the results of an empirical study on the rationale in IRC messages of OSS communities. We collected IRC logs from three OSS projects: Apache Lucene, Mozilla Thunderbird, and Ubuntu. We manually analyzed a sample of 7,500 messages and provide a quantitative evidence on how developers discuss rationale in their IRC messages. We found that committers contribute 54% of the discussed rationale on average. However, we did not find a strong correlation between the development activities and the rationale contribution. Finally, we evaluated various machine learning classification techniques for the automated extraction of rationale on two granularity levels: binary and fine-grained classification. We achieved 0.76 precision and 0.79 recall for the binary classification, and an average of 0.45 precision and 0.43 recall for the classification into finer-grained rationale elements. This study is a first step towards supporting the automated documentation of rationale in IRC messages.

Acknowledgments

We thank Ankur Sinha for his help in data collection. We also thank Jan Ole Johanßen and Sajjad Taheri for their valuable feedback. This work was partially supported by a PhD scholarship provided by King Saud University for Alkadhi.

References

[1] R. Alkadhi, J. O. Johanssen, E. Guzman, and B. Bruegge. REACT: An Approach for Capturing Rationale in Chat Messages. In *Proc. of the 11th International Symposium on Empirical Software Engineering and Measurement*, ESEM'17.

[2] R. Alkadhi, T. Laţa, E. Guzman, and B. Bruegge. Rationale in Development Chat Messages: An Exploratory Study. In *Proc. of the 14th International Conference on Mining Software Repositories*, MSR'17, pages 436–446, 2017.

[3] M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes. Automatic Extraction of Design Decisions From Issue Management Systems: a Machine Learning Based Approach. In *European Conference on Software Architecture*, pages 138–154, 2017.

[4] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining Email Social Networks. In *Proc. of the 2006 International Workshop on Mining Software Repositories*, MSR'06, pages 137–143, 2006.

[5] G. Breach. *I'm Not Chatting, I'm Innovating! Locating Lead Users in Open Source Software Communities*. University of Technology, Sydney School of Management, Working Paper Series, 2008.

[6] J. a. Brunet, G. C. Murphy, R. Terra, J. Figueiredo, and D. Serey. Do Developers Discuss Design? In *Proc. of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 340–343, 2014.

[7] M. Cataldo and J. D. Herbsleb. Communication Networks in Geographically Distributed Software Development. In *Proc. of the 2008 ACM Conference on Computer Supported Cooperative Work*, CSCW'08, pages 579–588, 2008.

[8] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.

[9] S. A. Chowdhury and A. Hindle. Mining StackOverflow to Filter out Off-topic IRC Discussion. In *Proc. of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 422–425, 2015.

[10] H. Cunningham, V. Tablan, A. Roberts, and K. Bontcheva. Getting More Out of Biomedical Documents with GATE's Full Lifecycle Open Source Text Analytics. *PLoS Comput Biol*, 9(2):e1002854, 2013.

[11] Y. Dittrich and R. Giuffrida. Exploring the Role of Instant Messaging in a Global Software Development Project. In *Proc. of the 6th IEEE International Conference on Global Software Engineering*, ICGSE'11, pages 103–112, 2011.

[12] C. Drummond and R. C. Holte. C4.5, Class Imbalance, and Cost Sensitivity: Why Under-sampling Beats Over-sampling. In *Workshop on learning from imbalanced datasets II*, volume 11, 2003.

[13] A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech. *Rationale Management in Software Engineering*. Springer-Verlag New York, Inc., 2006.

[14] M. S. Elliott. The virtual organizational culture of a free software development community. In *Proc. of the 3rd Workshop on Open Source Software Engineering*, 2003.

[15] M. S. Elliott and W. Scacchi. Free Software Developers As an Occupational Community: Resolving Conflicts and Fostering Collaboration. In *Proc. of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP'03, pages 21–30, 2003.

[16] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen. Communication in Open Source Software Development Mailing Lists. In *Proc. of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 277–286, 2013.

[17] M. Handel and J. D. Herbsleb. What is Chat Doing in the Workplace? In *Proc. of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW'02, pages 1–10, 2002.

[18] A. E. Hassan. The Road Ahead for Mining Software Repositories. In *Proc. of Frontiers of Software Maintenance*, FoSM'08, pages 48–57, 2008.

[19] A. Johri. Look Ma, No Email!: Blogs and IRC As Primary and Preferred Communication Tools in a Distributed Firm. In *Proc. of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW'11, pages 305–308, 2011.

[20] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proc. of the 29th International Conference on Software Engineering*, ICSE'07, pages 344–353, 2007.

[21] P. Kruchten, R. Capilla, and J. C. Dueñas. The Decision View's Role in Software Architecture Practice. *IEEE Softw.*, 26(2):36–42, 2009.

[22] W. Kunz and H. Rittel. Issues as Elements of Information Systems. Working Paper 131, Institute of Urban and Regional Development, University of California, 1970.

[23] Z. Kurtanović and W. Maalej. Mining user rationale from software reviews. In *Proc. of the 25th IEEE International Requirements Engineering Conference*, RE'17, 2017.

[24] J. Lee. Artificial Intelligence at MIT Expanding Frontiers. chapter SIBYL: A Qualitative Decision Management System, pages 104–133. MIT Press, 1990.

[25] T. C. Lethbridge, J. Singer, and A. Forward. How Software Engineers Use Documentation: The State of the Practice. *IEEE Softw.*, 20(6):35–39, 2003.

[26] Y. Liang, Y. Liu, C. K. Kwong, and W. B. Lee. Learning the "Whys": Discovering Design Rationale Using Text Mining - An Algorithm Perspective. *Comput. Aided Des.*, 44(10):916–930, 2012.

[27] A. MacLean, R. M. Young, V. M. E. Bellotti, and T. P. Moran. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction*, 6(3):201–250, 1991.

[28] R. McCall. PHIBIS: Procedurally Hierarchical Issue-based Information Systems. In *Proc. of the International Congress on Planning and Design Theory*, volume 44, 1987.

[29] M. McCandless and O. Gospodnetic. *Lucene in Action*. Manning Publications Co., 2005.

[30] K. L. Myers, N. B. Zumel, and P. Garcia. Automated Capture of Rationale for the Detailed Design Process. In *Proc. of the 16th National Conference on Artificial Intelligence and the 11th Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 876–883, 1999.

[31] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[32] K. A. Neuendorf. *The Content Analysis Guidebook*. Sage Publications, 2002.

[33] D. Pagano and W. Maalej. How Do Developers Blog?: An Exploratory Study. In *Proc. of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 123–132, 2011.

[34] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol. How Developers' Collaborations Identified from Different Sources Tell Us About Code Changes. In *Proc. 30th International Conference on Software Maintenance and Evolution*, ICSME'14, pages 251–260, 2014.

[35] L. Pascarella and A. Bacchelli. Classifying Code Comments in Java Open-source Software Systems. In *Proc. of the 14th International Conference on Mining Software Repositories*, MSR'17, pages 227–237, 2017.

[36] 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. Morenox, D. Shepherdxi, and E. Wong. On-Demand Developer Documentation. In *Proc. of Int'l. Conf. on Software Maintenance and Evolution*, 2017.

[37] G. Robles and J. M. Gonzalez-Barahona. Developer Identification Methods for Integrated Data from Various Sources. In *Proc. of the 2005 International Workshop on Mining Software Repositories*, MSR'05, pages 1–5, 2005.

[38] B. Rogers, J. Gung, Y. Qiao, and J. E. Burge. Exploring Techniques for Rationale Extraction From Existing Documents. In *Proc. of the 34th International Conference on Software Engineering*, ICSE'12, pages 1313–1316, 2012.

[39] B. Rogers, C. Justice, T. Mathur, and J. E. Burge. Generalizability of Document Features for Identifying Rationale. In *Proc. of the 7th International Conference on Design Computing and Cognition*, DCC'16, pages 633–651, 2016.

[40] B. Rogers, Y. Qiao, J. Gung, T. Mathur, and J. E. Burge. Using Text Mining Techniques to Extract Rationale from Existing Documentation. In *Proc. of the 6th International Conference on Design Computing and Cognition*, DCC'14, pages 457–474, 2014.

[41] R. R. Rosario. *A Data Augmentation Approach to Short Text Classification*. PhD thesis, University of California, Los Angeles, 2017.

[42] E. Shihab, Z. M. Jiang, and A. E. Hassan. On the Use of Internet Relay Chat (IRC) Meetings by Developers of the GNOME GTK+ Project. In *Proc. of the 6th IEEE International Working Conference on Mining Software Repositories*, MSR'09, pages 107–110, 2009.

[43] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the Use of Developer IRC Meetings in Open Source Projects. In *Proc. of the 2009 IEEE International Conference on Software Maintenance*, ICSM'09, pages 147–156, 2009.

[44] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky. The (R) Evolution of Social Media in Software Engineering. In *Proc. of the on Future of Software Engineering*, FOSE'14, pages 100–116, 2014.

[45] M.-A. Storey, A. Zagalsky, F. F. Filho, L. Singer, and D. M. German. How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. *IEEE Trans. Softw. Eng.*, 43(2):185–204, 2017.

[46] G. Tsoumakas and I. Katakis. Multi-label Classification: An Overview. *International Journal of Data Warehousing and Mining*, 3:1–13, 2007.

[47] E. Ukkonen. Algorithms for Approximate String Matching. *Inf. Control*, 64(1-3):100–118, 1985.

[48] L. Yu, S. Ramaswamy, A. Mishra, and D. Mishra. Communications in Global Software Development: An Empirical Study Using GTK+ OSS Repository. In *Proc. of the 2011th Confederated International Conference on On the Move to Meaningful Internet Systems*, OTM'11, pages 218–227, 2011.