How to Communicate when Submitting Patches: an Empirical Study of the Linux Kernel

XIN TAN, Peking University, China
MINGHUI ZHOU*, Peking University, China

Communication when submitting patches (CSP) is critical in software development, because ineffective communication wastes the time of developers and reviewers, and is even harmful to future product release and maintenance. In distributed software development, CSP is usually in the form of computer-mediated communication (CMC), which is a crucial topic concerned by the CSCW community. However, what to say and how to say in communication including CSP has been rarely studied. To bridge this knowledge gap and provide relevant guidance for developers to conduct CSP, in this study, we conducted an empirical study on the Linux kernel. We found four themes involving 17 expression elements that characterize what to express when submitting patches and three themes of contextual factors that determine how these elements are applied. Considering both expression elements and context, combined with an online survey, we obtained 17 practices for communication. Among them, four practices, such as "provide sufficient reasons" and "provide a trade-off" are the most important but difficult practices, for which we provide specific instructions. We also found that the "individual factors" plays a special role in communication, which may lead to potential problems even in accepted patches. Based on these findings, we discuss the recommendations for different practitioners, including patch submitters, reviewers, and tool designers, and the implications for open source software (OSS) communities and the CSCW researchers.

CCS Concepts: • **Human-centered computing → Collaborative and social computing theory, concepts and paradigms**; Social media.

Additional Key Words and Phrases: Communication Content; Patch Submission; CMC; Linux Kernel

ACM Reference Format:
Xin Tan and Minghui Zhou. 2019. How to Communicate when Submitting Patches: an Empirical Study of the Linux Kernel. In *Proceedings of the ACM on Human-Computer Interaction*, Vol. 3, No. CSCW, Article 108, November 2019, ACM, New York, NY, USA. 26 pages. https://doi.org/10.1145/3359210

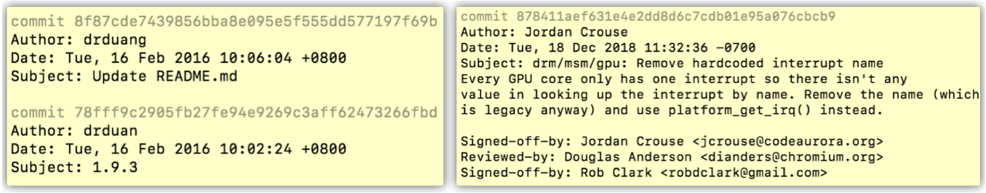
1 INTRODUCTION

Communication is essential in software development, which takes developers substantial time [3]. In distributed development, computer-mediated communication (CMC) is pretty common but more challenging because of the reduced opportunities for team members to meet face-to-face, coupled with different time zones and cultural differences [37, 45]. Considerable effort has been made on optimizing CMC by researchers from both the software engineering community [43, 46, 55] and the

*Corresponding author

Authors' addresses: Xin Tan, Peking University, Department of Computer Science, School of EECS, Key Laboratory of High Confidence, No.5 Yiheyuan Road, Haidian District, Beijing, China, 100871, tanxin16@pku.edu.cn; Minghui Zhou, Peking University, School of Electronics Engineering and Computer Science, Key Laboratory of High Confidence, No.5 Yiheyuan Road, Haidian District, Beijing, China, 100871, zhmh@pku.edu.cn.

2573-0142/2019/11-ART108 $15.00
https://doi.org/10.1145/3359210



```
commit 8f87cde7439856bba8e095e5f555dd577197f69b
Author: drduang
Date: Tue, 16 Feb 2016 10:06:04 +0800
Subject: Update README.md

commit 78fff9c2905fb27fe94e9269c3aff62473266fbd
Author: drduan
Date: Tue, 16 Feb 2016 10:02:24 +0800
Subject: 1.9.3
```

```
commit 878411aef631e4e2dd8d6c7cdb01e95a076cbcb9
Author: Jordan Crouse
Date: Tue, 18 Dec 2018 11:32:36 -0700
Subject: drm/msm/gpu: Remove hardcoded interrupt name
Every GPU core only has one interrupt so there isn't any
value in looking up the interrupt by name. Remove the name (which
is legacy anyway) and use platform_get_irq() instead.

Signed-off-by: Jordan Crouse <jcrouse@codeaurora.org>
Reviewed-by: Douglas Anderson <dianders@chromium.org>
Signed-off-by: Rob Clark <robdclark@gmail.com>
```

Fig. 1. Examples of Commit Messages from an OSS Project and from the Linux Kernel

CSCW community [29, 41, 52]. However, few studies focus on the content of communication and guiding developers what/how to say when performing certain tasks. This might be because human expression is complicated by nature, e.g., uncertainty in communication context and differences of communicators' characteristics [8]. In this paper, we bridge the gap by studying **what/how to say when submitting patches (RQ)**, which is a critical problem in software development.

Text is the only means to communicate when developers submit patches through either email or pull-request [69]. Developers should include natural language descriptions along with patches to facilitate code review. For example, in the Linux kernel, developers use emails to submit and review patches. In a new release, there are 8,000 to 12,000 patches integrated, which are contributed by 1,000 developers [17]. These patches only represent the "lucky few." Studies have shown that only around 33% of patches are accepted [33]. Among the rejected patches, many are even neglected by the reviewers owing to the flood of massive patches in the mailing list [47]. Even if the patches successfully attract attention, many of them are still rejected owing to inappropriate descriptions. Given the great number of patches and the limited energy of reviewers [67], how to catch the reviewers' attention and make communication more effective are very important.

Communication when submitting patches (CSP) is also closely related to software maintenance [32]. Once patches are accepted and applied, their descriptions will be stored in repositories as permanent records until projects are abandoned. Large-scale OSS projects, particularly that serve as critical computing infrastructure for our society (e.g., operating system), usually need to be maintained for a few decades [61]. When developers go away, the natural language descriptions of patches are often the only information they leave for next-generation developers to understand these modifications. Fig 1 shows two examples of poor CSP from an OSS project and one example of good CSP from the Linux kernel. The two examples on the left lack valid information (e.g., motivation), which may bring serious problems for maintenance, while the example on the right provides valuable references for developers to understand the modification.[1] Furthermore, CSP plays an important role in the release note generation, because accurate descriptions of patches provide a reliable data source for creating/generating release note [66].

By better understanding the practices for CSP, we can provide policies and tools that enable developers to achieve better communication, which has profound significance. Considering that the Linux kernel is one of the most prominent open source projects, and it has accumulated abundant mature experiences during its nearly 30-year history [42], we chose it as the case and conducted an empirical study. By analyzing the emails for communicating patches combined with an online survey, we obtained 17 practices, which involve two themes of contextual factors (i.e., **community rules** and **patch factors**) and four themes of message descriptions (i.e., **timing of CSP**, **granularity of CSP**, **content of CSP**, and **manner of CSP**). Among them, four practices including **#6. include technical details**, **#8. provide a trade-off**, **#13. provide sufficient reasons**, and **#15. provide the way to reproduce the bug** are considered to be the most important yet difficult ones, which

[1]It does not mean that the more information the better.

require attention from patch submitters in practice. We found that **individual factors**, the third theme of contextual factors, plays a special role in CSP, which means if submitters are with authority or high reputation, or modifications of patches are urgently needed by reviewers, the descriptions of patches sometimes may omit important information. This type of patches can also be accepted as long as the code is correct. However, this may lead to potential problems with software maintenance. In summary, the major contributions of our study are: (1) a comprehensive list of **expression elements** that characterize what to say when submitting patches; (2) an in-depth empirical investigation of the **contextual factors** that influence what to say, i.e., why to say it; (3) **practices for effective communication** that tell how to say when submitting patches; (4) **recommendations** for patch submitters, reviewers, and tool designers, and **implications** for OSS communities and CSCW researchers.

The remainder of this paper is organized as follows. In Section 2, we present the background. We introduce the dataset and the methods we used in Section 3. We report and validate our findings in Sections 4 and 5, respectively, followed by a discussion in Section 6. We outline the threats to validity in Section 7 and, finally, conclude the paper in Section 8.

2 BACKGROUND

Software development as a core CSCW domain is essentially a collaboration between multiple software developers and related other professional groups [4]. CSCW's interest in the area of software development is about understanding the complexity of coordinating cooperative activities aiming to develop various types of technologies and IT systems in order to decrease the complexity of coordination and articulation work [50]. In the process of software development, communication is an important manifestation and way of collaboration [30] and has attracted extensive attention from both software engineering community and CSCW community. To provide a background for understanding our study, we summarize literature on CMC and contribution acceptance in software development.

2.1 Media of CMC

While geographically distributed development becomes pervasive on modern software projects [39], CMC achieves collaboration and task awareness among developers [7, 28, 54]. At the same time, compared to face-to-face communication, CMC brings challenges because of its limitations, e.g., responsiveness [38] and miscommunication on text-based channels [55]. The previous studies revolve around the media of CMC, mainly involving the following two aspects. One of the focuses is on understanding the characteristics of media for CMC in distributed virtual teams. In particular, the members of virtual teams can either be in real time as in the case of a video conference, instant messaging [63] and chat [31, 58], or in an asynchronous setting such as email [60]. Because synchronous means, e.g., video conference, are more difficult to arrange when teams work in different time zones, developers usually prefer to use asynchronous means [15]. Through a study on three well-known OSS projects, Gutwin et al. found that text-based communication is the main way to communicate [29], which is the form of CSP studied in this paper. The idea of Social Mechanism of Interaction introduced by Schmidt, emphasized the role of product itself (e.g., source code) in supporting the articulation of the distributed activities of multiple actors [49]. Not only code base [22], but also bug report forms [5, 49] are means by which the articulation work of the project can be carried out. Similarly, good software documentation was found to be a useful reference point for developers to communicate their ideas and intentions to one another, which requires various communication skills that go far beyond the skills required to fix a bug or write a new feature [25].

Another focus is on investigating how developers use these media in practice. Based on a large-scale survey, Storey et al. [54] investigated how the choice of different communication media shaped developers' activities within a participatory culture of development. They found that code hosting sites were considered the most important communication media for team collaboration, group awareness, and project coordination. An example of improved quality of communications is described by Cataldo et al. [12] where the most productive developers, changed their use of communication media over time, achieving higher congruence between coordination requirements and activities. Advanced tools have been proposed to foster efficient CMC. For example, Angelo et al. [20] designed a novel gaze visualization for the communication among remote pair programmers. Some tools were designed to help developers ask questions, e.g., Expertise Recommender [41], Expertise Browser [44], and Yoda [11].

2.2 Content of CMC

While substantial effort has been made around the media of CMC, few studies focus on the communication content. Fortunately, because the information in bug reports influences the speed at which bugs are fixed [2], some researchers have paid attention to what information is needed in a bug report. For example, Ko et al. [36] conducted a study about how people describe software problem. Through syntactic analysis, they analyzed titles of bug reports from five popular projects, and obtained the top ten words for each part of speech including nouns, verbs, adjectives, conjunctions and prepositions. Information such as steps to reproduce, stack traces, and test cases in bug reports are considered helpful by developers, which are, at the same time, most difficult to provide for users [6]. Because bug tracking systems play a central role in supporting collaboration between the developers and the users of the software, how to write a good bug report also have attracted CSCW researchers' attention. Breu et al. [9] analyzed 600 bug reports in the repositories of two large OSS projects, and found that users' active and ongoing participation is important for making progress on the bugs they report. They also found that information such as screenshots, stack traces, or steps to reproduce, are important. In a more recent study [14], by manually analyzing nearly 3k bug reports, researchers found that while most reports (93.5%) contain *Observed Behavior*, only 35.2% and 51.4% explicitly describe *Expected Behavior* and *Steps to Reproduce*, respectively. They also designed and evaluated an automated approach to detect the absence of these information.

2.3 Contribution Acceptance

Our work is also related to contribution acceptance in software development, especially from the perspective of communication. Many studies investigate the factors that influence contribution acceptance (either patch based or pull-request based). For example, Gousios et al. identified *quality* and *prioritization* are the two key factors that integrators are concerned with [28], whereas from the perspective of contributors, challenges are mostly social in nature, for example, contributors feel that communication is the main challenge when submitting pull-request [27]. Based on the manual inspection of 300 rejected Eclipse and Mozilla patches, a large-scale online survey, and the literature review, the researchers found that besides the technical problems of implementation, bad timing of patch submission and a lack of communication with team members can also result in a negative patch review [57]. Steinmacher et al. [53] found that communication is one of the main social barriers when newcomers submit code because they feel difficult in sending a meaningful/correct message, which is exactly the question addressed by our study. The factors of gender bias [59], submitters' experience [33], number of comments, and the size of a pull request [65] are also considered the indicators that influence contribution acceptance.

Although both developers and researchers have recognized that, when submitting patches, communication is important and challenging, what/how to say has been rarely studied. A relevant study conducted by Tsay et al. [62] just focused on the content of the discussion around pull-requests. They found that developers raised issues around contributions over both the appropriateness of the problem that the submitter attempted to solve and the correctness of the implemented solution. However, they did not look at how to describe patches, which is the base of collaboration between developers and reviewers (a core research interest for CSCW) and is considered a critical but difficult process by developers [27], particularly newcomers [52]. In this study, we try to address this issue by an empirical study of the Linux kernel.

3 METHODS

We collected and analyzed online documents and emails of submitting patches to understand CSP in the Linux kernel. By coding the texts of emails and documents, we extracted the key information to characterize CSP. On the basis of the information obtained, we reanalyzed the emails of accepted patches and combined with an online survey to obtain practices and insights to help developers to communicate effectively and wisely. The remainder of this section introduces how we collected data, extracted key information of CSP, including **expression elements** and **contextual factors**, and how the practices were obtained.

3.1 Collection and Preparation of Data

3.1.1 Selecting Documents. We read the online documents to understand the Linux kernel community's rules on patch submission. We first collected related documents by using Google to search for the keywords "patch submission" and "Linux kernel," and used the top 50 most relevant results as our initial dataset. We then expanded these documents by retrieving the embedded links. We iterated this process until no new document emerged. Eventually, we identified 23 documents that were related to the rules or official instructions of patch submission, among which 15 documents referred to how to describe patch, but some were just a few words.

3.1.2 Selecting Patches. In the Linux kernel community, there is one global mailing list, called the *Linux kernel Mailing List* (LKML), and 130 more specialized lists. In general, LKML contains all the patches because the kernel community stipulates that the patches sent to those specialized lists need to be copied to LKML [18]. To avoid obtaining duplicate patches, we only focus on LKML. We utilized the data provided by Xu and Zhou [64], which shares a multi-level dataset of the Linux kernel patchwork[2] covering a nine-year history (Dec. 2008 to Dec. 2017) of more than 666,550 patches and related discussions recorded by LKML. Among all the discussions, we mainly focused on the descriptions that developers attached at submission time. To understand what the effective CSP is, we need to know which patches were accepted and which ones were rejected, and further to know which ones were rejected because of poor communication. By relating a patch (in LKML) to a commit (in the code repository), this dataset had already distinguished between accepted patches and rejected ones.

It was not easy to identify the patches that were rejected because of inappropriate communicating expression. To attain the goal, we exploited the patches that have multiple versions but only differ in their description (11,911 patches). These patches got rejected in the beginning and later got accepted with modified description (and with the same code implementation). For this kind of patches, we were able to confirm that the rejection of the initial versions was due to communication problems. For other rejected patches, it was hard to know they were rejected because of description or simply implementation. Specifically, we did the following steps. We used MD5 hash function [48] to compress the descriptions and code for all the patches. For each accepted patch, we looked for its counterparts in the rejected patches that have the same code but different descriptions based on the hash values. The found patches were the initial versions of the accepted patches that have exactly the same code. We considered them rejected because of poor communication. However,

[2]https://lore.kernel.org/patchwork/project/lkml/list/

this approach may bring threats when the requirements changes (the details are presented in Section 7.1). In order to alleviate this threat, we restrict the time interval between two versions to be less than six months, because the review time is usually no more than two development cycles (each cycle lasts two to three months) [33]. We randomly selected 50 sets of these patches for manual check. We compared the descriptions of different versions and read their comments to understand why they were rejected. We found that all these patches were indeed rejected because of communication, which verified the reliability of our approach.

By exploring all the 139,664 accepted patches, we found that 11,911 (8.5%) patches had been iterated more than two times because of ineffective communication (# versions per patch: median: 2.0; mean: 2.4). The review time (from the sending time of the first version to the time it is accepted) of these patches is 36.4 days on average (median: 20.0 days), whereas for the patches whose first versions were accepted, the review time is relatively short (median: 4 days; mean: 14.1 days). It indicates that ineffective CSP is a pervasive and serious problem.

To understand CSP, we randomly selected 700 patches for manual analysis in the next steps, including 500 patches among 139,664 accepted patches (confidence level: 95%, margin of error: 4.37%)[3] and 200 patches among 11,911 patches that were rejected because of communication problems (confidence level: 95%, margin of error: 6.87%). The process of extracting the key information of CSP involved 500 accepted patches. The process of identifying practices involved 500 accepted patches in the previous step, and another 200 rejected ones for further exploration. The reason for choosing 700 patches was that more patches were not likely to provide new information. In fact, when we finished analyzing the 23 online documents collected in Section 3.1.1 and 150 accepted patches, all the codes had already emerged. It means that saturation has been reached, which is a common approach to mulling over sample size decisions in qualitative research [26, 40].

3.2 Extraction of the Key Information of CSP

The message, the context, and the medium are the essential factors of communication [23]. The effectiveness of the communication of a message is partly determined by an astute choice of the medium and context in which to deliver them [24]. For patch submission in the Linux kernel, the medium is fixed: email. Thus, it is important to explore how a message is expressed and what the context is. To analyze the message (combination of elements) and context of CSP, we applied thematic analysis, a method that is often used in qualitative analysis [19]. To limit subjectiveness, for all manual analysis in this section, two authors engaged individually (e.g., inductive open coding). We then compared our list of codes and themes, and developed a coding guide with definitions and examples for each identified theme. Examples from initial coding guide are shown in Table 1. Each author then used the coding guide to independently analyze the complete set of data. Based on a discussion of the second round of analysis, the coding guide was further refined, and the data were independently analyzed for a third and final time. The final inter-rater reliability was 0.96 (Cohen's kappa), and discrepancies were discussed and resolved by introducing a colleague, who has rich experience of qualitative analysis and is familiar with our research, as an arbitrator [21].

The remainder of this section describes how we used thematic analysis to extract expression elements and contextual factors in detail, respectively.

3.2.1 Extraction of the Expression Elements. We followed the method of thematic analysis [19], which mainly included the following steps. (1) We first read and reread all of the online documents and the descriptions of 500 randomly selected accepted patches to understand how developers communicate when submitting patches. (2) Generate initial codes. In this phase, we start to organize our data in a meaningful and systematic way. For each document we analyzed, we identified the

[3]https://www.qualtrics.com/blog/calculating-sample-size/

Table 1. Examples from Initial Coding Guide

Codes from individual	Definition	Code	Theme
Failed: Device or resource busy (*author1*) Failed to lock logical volume vg0/lvol_rootfs (*author1*) Fix below issues reported by checkpatch (*author2*)	Statements on how to reproduce bug, e.g., providing executing method, error message.	Bug reproduction	Description of problem
Causes asynchronous external abort (*author1*) This will cause memory leak (*author1*) This leads to crash signatures (*author2*)	Statements on serious impacts caused by bugs, e.g., memory leak	Seriousness of bug	
The First bad commit is ... (*author2*) Related commit is ... (*author2*)	Statements on related commits of a bug.	Related commit	

community rules for patch submission, as well as the suggestions that help developers get their patch accepted, then conducted open coding on these examples; for the descriptions of 500 accepted patches, we carefully read the sentences and generated the initial codes that characterized the key information contained. (3) We grouped the codes into initial themes that were conceptually similar. At the end of this step the codes had been organized into broader themes that seemed to explain which information needs to be included and what should be paid attention to in CSP. (4) We reviewed and modified the initial themes and found opportunities for merging. (5) Then, we defined the final themes aiming to identifying the "essence" of what each theme is about, for example considering whether there were sub-themes and how they interacted and related to the main theme. For this process, we refer to the **sub-theme** we obtained as **element** because the difference in the manner of communication is precisely reflected in the combination of these elements.

3.2.2 Extraction of the Contextual Factors. We found that the frequency of the different elements was fairly unbalanced, e.g., 93.3% of the accepted patches had **motivation**, whereas 8.0% of the accepted patches had **related commit**. Therefore, we realized that the elements that may appear in the communication were related to the context. To understand what the communication context was, we explored the 500 accepted patches with annotated elements analyzed in Section 3.2.1. For instance, we analyzed the specific scenario of each patch including what elements were and were not included and annotated the reasons as to why some were included and others were not. We annotated the reasons by considering the characteristics of the patch and its context, e.g., the number of lines of patches, the type of patches, the community rules, and the discussions in the same threads. For a small number of patches that we could not know the reasons behind their descriptions, we emailed the authors to ask why they described them in these ways (there are nine cases, and for six of them, we got replies; for the other three we did not get reply, we emailed their reviewers and obtained some insights). Through this step, we obtained the data related to the different contexts. Then, we processed this part of the data using steps similar to those we used to extract expression elements. Eventually, we obtained five sub-themes belonging to three themes to describe the context of CSP. We call the **sub-themes** we obtained as **contextual factors**, because the communication (how to use these expression elements) is influenced by the five factors, each of which has many different values.

3.3 Extraction of the Practices

3.3.1 Extraction of the Candidate Practices. After we obtained the expression elements and contextual factors of CSP, we could identify how to combine these elements in certain context by observing the communication of the accepted patches, because their communication is considered effective. We utilized the previous 500 accepted patches to explore the elements they had and, at the same time, fully considered the context of patch submission to locate candidate practices. For example, we found that among the descriptions of 240 bug fixing patches, ten of which only contained the element **motivation** without **bug reproduction**. By analyzing their contextual factors, we found that these patches were all trivial fixes, e.g., typo fix. Therefore, we summarized "*(f)or patches that are obvious or trivial, just indicate what you did.*" as one candidate practice for CSP. Eventually, we obtained 18 candidate practices. Through a developer survey, we kept 17 practices. We also randomly selected 200 patches that were rejected due to inappropriate communication to analyze their compliance with practices.

3.3.2 Developer Survey. To validate the candidate practices and locate the most eminent practices, we conducted a survey with kernel developers. The survey had two parts. In the first part, we inquired about their experiences with CSP to understand their difficulties in CSP. In the second part, we asked the participants about their views toward all the candidate practices. Among them, 16 could be easily identified as good practices because they provided valuable information that can save reviewer effort. However, for the remaining two, such as "omit some information in certain context to save time," we could hardly assess whether they were effective. For the practices that could be easily identified, we asked the developers to score the importance and operating difficulty with a five-point Likert scale, and for the other two we asked what they thought of them. We also asked developers to suggest additional practices. However, we did not receive any new practice, which suggests that the practices we obtained are relatively comprehensive. To screen out valuable practices and obtain effective feedback, we sent the emails containing the link of our online survey to 200 active and experienced kernel developers, the standard being that they contributed within the last year and participated at least three versions (confidence level: 95%, margin of error 6.12%). The reason why we chose active and experienced kernel developers is because we wanted to screen out valuable practices and obtain effective feedback. Because most of these developers had experiences of being newcomers when they joined, it is reasonable to ask their experiences with CSP to understand the difficulties they had gone through. On the contrary, choosing newcomers is hard to obtain these informative answers. The survey ran for two weeks. Finally, we received 26 valid responses (response rate: 13%). This rate is comparable to the 6% to 36% reported for other surveys in software engineering [51]. The survey can be found in our online appendix.[4]

4 FINDINGS

In this section, we present our findings including **expression elements** (what information may be contained in CSP), **contextual factors** (what factors may influence whether or not to contain these expression elements), and **practices of CSP** (in the certain context, what expression elements should be used to make CSP effective) that we discovered from the Linux kernel. To enable traceability, we include direct quotes from emails and documents identified in our dataset.

4.1 Expression Elements of CSP

We retrieved 17 elements that fall into four major themes to characterize the message expression of patch submission from documents and emails, as shown in Table 2. Except for **timing of CSP** that we extracted from only one official document of the Linux kernel, all of the other themes and elements were extracted from multiple documents or emails, which ensures that these elements are not accidental. Below are the explanation of these four themes, along with the excerpts from their sources.

[4]https://zenodo.org/record/3365612

Table 2. Themes and Expression Elements of CSP (The column of "Document" represents the index of the online documents in the appendix.[4]; "✓" in the column of "Email" represents the elements were extracted from multiple emails.)

Themes		**Elements**	**Explanation of the Elements**	**Extraction Source**	
				Document	**Email**
Timing of CSP		Timing of submission	Timingto submit patches.	#13	
Granularity of CSP		Separation of change	Requirement for patch granularity.	#1, #10, #13	
Content of CSP	Description of motivation	Motivation	Reason for developer's actions, which usually contain why s/he did and what s/he did.	#1, #2, #6, #8, #13	✓
	Description of problem	Bug reproduction	Scenario of bugs, ways to produce it, or IDs of bugs which have been reported.	#1	✓
		Seriousness of bug	Influence of problems, such as "crash," "warning."	#1	✓
		Related commit	Sometimes a bug is related to a commit. Information of commit ID and summary.	#1	✓
	Description of implementat ion	Technical details	Description of how to implement it.	#1, #2	✓
		Brief introduction of functions	Description of functions contained in new features.		✓
		Improvement	Such as "reducing time/memory," "making code more readable," etc.	#1	✓
		Trade-off	Other side effects when applying patches.	#1	✓
		Reference	References to other resources, such as emails, other patches, or experts.	#2	✓
	Description of quality	Superiority	Superiority of solutions or information about uniqueness.		✓
		Test/review	Information of quality, such as "Tested-by".	#1, #2, #7	✓
		Limitation	Limitations of patches, such as only apply to the certain situation, have not implemented all the functions.		✓
Manner of CSP	Language	Mood of speaking	Mood of CSP, such as using the imperative mood.	#1, #2	
	Satisfy basic requirement of community	Basic requirement of subject	Including begin with "[patch]," "serial number padding 0 if it has," "file location information," and "brief description of the modification."	#1 #2, #3, #4, #5, #8, #12, #15	✓
		Width of description	Width of the message. (75 columns)	#9, #11, #12	

4.1.1 Timing of CSP. For any communication, finding a right time to start the talk is important. If both parties are busy with something or simply not in the mood to communicate, the talk will not go smoothly. For example, too fast or too slow response suggests the lack of attention from the community and decreases the odds of a newcomer becoming a long term contributor [68]. This is particularly important with CSP:

> *Publishing a patch is a very important event in its lifetime. The timing of the submission is closely related to whether it can be merged into the new release. There is a delicate balance between posting too early and posting too late. (DOC #13)*

4.1.2 Granularity of CSP. There are a huge number of patches submitted to LKML every day. To review patches efficiently, the community prioritizes condensed patches. It is fairly important to submit larger changes in smaller pieces so that the reviewers and maintainers can process the changes step by step. This theme has two facets: the granularity of a single patch and the way of submitting large patch kits:

> *NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people! Condense your changes into more manageable pieces or only submit smaller groups of changes at a time. (DOC #10)*
> *For example, if your changes include both bug fixes and performance enhancements for a single driver, separate those changes into two or more patches. If you cannot condense your patch set into a smaller set of patches, then only post say 15 patches or so at a time and wait for review and integration. (DOC #1)*

4.1.3 Content of CSP. To convey the information that the submitter wants, and to satisfy the expectations of the reviewers, is the most important yet difficult thing about CSP. This theme involves the main content of CSP. The expression elements belonging to this theme can help developers to express the appropriate content in a specific scenario. As shown in Table 2, unlike the two themes mentioned above, which contains only one expression element each, the elements under this theme are very rich. We found that the following four sub-themes usually appeared in the content of CSP.

The sub-theme **description of motivation** contains only one expression element: **motivation** that describes why developers submit patches. This information is essential for reviewers to understand the necessity of patches. It usually appears in the first sentence of the content of CSP and almost all the patches have this element, for example:

> *Currently the tools/vm Makefile has a rather arbitrary implicit build rule; page-types is the first value in TARGETS so lets just build that one! Additionally there is no install rule and this is needed for make -C tools vm_install to work properly. (Email #750879)*[5]

The sub-theme **description of problem** also appears frequently in the content of CSP, but it only appears in patches of bug fixes. As shown in Table 2, three expression elements including **bug reproduction**, **seriousness of bug**, and **related commit** belong to this sub-theme. These expression elements are intended to help reviewers confirm that a patch fixes the real or even serious problem. The following three examples show the extraction sources of these three elements.

> **bug reproduction** — *Here is the backtrace of the bug [0.771174] OF: PCI: MEM 0x60000000.. 0x6fffffff -> 0x60000000 (Email #747511)*
> **seriousness of bug** — *Recent fixes for iATU unroll support introduced a bug that causes asynchronous external abort in Keystone PCIe h/w which doesn't have ATU port and the corresponding register. (Email #747511)*
> **related commit** — *If your patch fixes a bug in a specific commit, e.g. you found an issue using "git bisect", please use the 'Fixes:' tag with the first 12 characters of the SHA-1 ID, and the one line summary. (DOC #1)*

Another key information that appears a lot in the content of CSP is **description of implementation**. It can help reviewers to judge whether the implementation of a patch is reasonable or not. For different scenarios, we found the following five expression elements may be used to describe the implementation of a patch: **technical details**, **brief introduction of functions**, **improvement**, **trade-off**, and **reference**. Table 2 provides the explanation of these elements. The following examples show the extraction sources of **improvement** and **reference**.

> **improvement** — *Even in the signed case this gives us room for 8PiB (pebibytes) in size whilst still allowing the usual negative value error checking idiom. (Email #718954)*
> **reference** — *As pointed out by Ray Jui, there is a related problem... (Email #708330)*

[5]ID of a patch in Patchwork.

Patches with high quality are usually welcomed by reviewers. The community of the Linux kernel requires all the patches to be signed by the relevant responsible developers, which can effectively guarantee and track the quality of patches. By analyzing the description of patches, we also found that some patches contain sentences that indicate the quality of patches, e.g., advantages and disadvantages of current implementations. For all this information, we summarized it as **description of quality**, which is another sub-theme belonging to the theme **content of CSP**. There are three expression elements under this sub-theme. The following examples show the extraction sources of these three elements.

superiority — *This algorithm is more efficient than calling regular(). (Email #812387)*
test/review — *If your role was purely review and/or testing, please use Acked-by: instead. (DOC #2)*
limitation — *side effect: more memory (Email #779409)*

4.1.4 Manner of CSP. The expression elements belonging to this theme involve all the requirements from the community of patch submission, including **the language and the format**:

language — *"make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz," as if you are giving orders to the codebase to change its behaviour. (DOC #1)*
format — *Prefix the subject line with "[PATCH]" owing to the high email traffic to the Linux kernel community — [Patch] PCI: designware: Check for iATU unroll support after initializing host (Email #9377559)*

4.2 Contextual Factors of CSP

In Section 4.1, we have identified all the expression elements that may be used in CSP. However, not all these elements should be simultaneously used when submitting a patch. When extracting expression elements, we found the frequencies of these elements to be rather different. For example, all the patches we sampled satisfied **width of description** (the width of message is 75 columns), whereas only two patches contained **trade-off**. Therefore, we wanted to understand what factors may influence the presence or absence of these elements. To achieve this goal, we explored 500 accepted patches with annotated elements analyzed in the element-extraction step. For each of them, we carefully analyzed the reasons why each element appeared or did not appear and annotated them with reasons (see Section 3.2.2 for details).[6] By classifying and abstracting these reasons (see Table 3), we found five factors that belong to three themes related to the context of CSP.

4.2.1 Rules of the Community. It involves only one contextual factor: **rules of the community**. The reason for emphasizing this factor is that we found that many elements involved in the communication are directly decided by the particular rules of the Linux kernel community. As shown in Table 3, for the elements that belong to the following themes **timing of CSP**, **granularity of CSP**, and **manner of CSP**, the kernel community has clearly stipulated how to express these elements in its contributing guide (refer to the elements marked "R"). In other OSS communities, the expression of these elements may be different. For example, in the PostgreSQL community, developers are required to pack regression tests together when submitting a patch [16]. However, in the Linux kernel community, the pre-commit testing is usually based on a development version of the mainline repository or one of the other unstable/development trees for a related project. Therefore, the community stipulates that developers should organize each logical change as a

[6] Note that the reasons we annotated are the most likely or straightforward causes, but we can not guarantee that they are completely correct. For example, if the description of a patch did not mention its limitation, we annotated the reason was that the patch has no limitation, but in fact it may because that the submitter was unaware of its limitation. This is something that we cannot identify.

Table 3. Reasons that are Associated with the Occurrence of the Expression Elements (The numbers in the parentheses represent the number of patches which have or don't have the specific element because of that reason; The alphabets in square brackets represent the contextual factors that was extracted from these reasons: **T**: type of the patch; **R**: rules of the community; **S**: submitter's characteristics; **E**: reviewer's expectation; **I**: implementation of the patch.)

Elements	**Reasons We Annotated**	
	Reasons for CSP containing this element	**Reasons for CSP not containing this element**
Timing of submission	Community's requirement: Only patches of bug fix can be merged in merged window. Others should be submitted outside the window. **(500) [R, T]**	None
Separation of change	It is the requirement of the community. **(484) [R]**	The subsidiary modifications are simple, e.g., fix a typo. **(16) [T, I]**
Motivation	Almost all the patches need it to explain "why" and "what". **(466) [T]**	The motivation is very obvious and modification is simple; **(20) [T, I]** The author is the maintainer; **(10) [S]** The feature has been discussed. **(4) [E]**
Bug reproduce	This patch fixes a bug. **(230) [T]**	The patch is not a bug fix. **(260) [T]** The bug is very trivial and the subject has already explained what the bug is. **(10) [I]**
Seriousness of the bug	This patch fixes a bug and the bug is serious. **(154) [T]**	It is not a bug fix; **(260) [T]** The bug is not serious. **(86) [T]**
Related commit	This patch fixes a bug and it is related to a commit. **(41) [T, I]**	It is not a bug fix; **(260) [T]** It is not related to a commit; **(199) [T]**
Technical details	The implementation is complex. **(266) [I]**	The implementation is easy; **(204) [I]** The author has the certain right (maintainer); **(10) [S]** The feature has been discussed. **(20) [E]**
Brief introduction of functions	This patch adds a new feature. **(34) [T]**	The patch does not add a new feature; **(456) [T]** The author is the maintainer of this modules. **(10) [S]**
Improvement	After applying the patches, there is an improvement, e.g., clean up the code, simply the code. **(130) [T, I]**	Maybe there is no improvement after applying the patches. **(370) [I]**
Trade-off	The implementation has a trade-off. **(4) [I]**	The implementation does not have trade-off or because other personal reasons, the author does not want to tell the trade-off. **(496) [I]**
Reference	The developer refers to the resources to implement the patch. **(70) [I]**	The developer may not refer to the resources to implement the patch. **(430) [I]**
Superiority	There is more than one way to implement the patch. **(12) [I]**	Maybe there is only one way to implement or there are several ways that achieve the same effect. **(488) [I]**
Test/review	It is tested by others, usually for complex patches. **(116) [I]**	It is not tested by others. **(384) [I]**
Limitation	The patch is initial version or prototype. **(10) [I]**	Maybe the patch does not have limitation. **(490) [I]**
Mood of speaking (imperative mood)	It is the strong suggestion of the community. **(436) [R]**	The reviewer looks forward to this type of patch very much, e.g., bug fix; **(36) [E]** The implementation is vey complex, so the language becomes a secondary factor. **(28) [I]**
Basic requirement of subject	It is the requirement of the community. **(494) [R]**	The author is experienced and has a certain right. **(6) [S]**
Width of description	It is the requirement of the community. **(500) [R]**	None

separate patch and include the tags such as "Tested-by" in the descriptions of patches to indicate that the patches have been successfully tested. This theme decides the basic requirement for mail interaction of a patch submission, which is crucial to the acceptance of patches.

4.2.2 Patch Factors. What to say when submitting patches is directly decided by the nature of patches. For example, why the message contains **bug reproduction**? Because the type of the patch is fixing a bug, and this information can help the reviewer to confirm whether it is a real problem (see line 5 of Table 3). There are two contextual factors included in this theme: **type of the patch** and **implementation of the patch**. The expression elements marked “T” or “I” in Table 3 are influenced by these two factors. Although it seems that only two factors are involved, the number of the specific contexts formed is possibly very large.

By analyzing the 500 accepted patches, we discovered 11 types of patches, including **adding core function**, **adding driver**, **code refactoring**, **fixing bug**, **easy case**, etc. Furthermore, we observed the distribution of the 17 elements on different types of patches. We found some rules. For example, almost all the types of patches satisfied the **basic requirement of the community**; the patches of **adding core function** usually included **technical details**, while it may be unnecessary for **easy case**; some elements only appear in certain types of patches, e.g., **bug reproduction**, **seriousness of the bug**, and **improvement**; **easy case** and **adding driver** usually omitted the “why” and just mentioned the “what” to express the motivation. These suggest that the usage of the expression elements is related to the **type of the patch**, e.g., for **easy case**, one or two sentences may be enough to make the expression clear, whereas if developers want to submit a complex feature, not only should they provide sufficient reasons, but they also need to describe how it is implemented. Therefore, developers should pay special attention to the corresponding expression elements of the specific patch types.

Developers also need to consider **implementation of the patch**. It is the most complex contextual factor, which includes the implementation details of the patch: its **implementation approach**, **difficulty**, **maturity**, **application effect**, etc. For example, if the developer refers to other resources, s/he needs to provide the reference: *I tried to mark the exact input buffer as an output here, but couldn't figure it out. As suggested by Linus, marking all memory as clobbered however is good enough too (Email #846653).* If there is a side effect, developers should quantify the optimization and trade-offs so that the reviewer can weigh the costs against the benefits.

4.2.3 Individual Factors. Patch submission is a process of communication between developers and reviewers. It is essentially the same as everyday communication, both in terms of transmission and exchange of information. A good communication should express the meaning faithfully, while meeting the expectation of the receiver. Thus, we considered the **individual factors** as one of the important themes that influence the expression. It includes two contextual factors: **submitter's characteristics** and **reviewer's expectation**. All the elements, except those related to the basic requirement of the community, are influenced by this theme (refer to the elements marked “S” or “E” in Table 3).

The contextual factor **submitter's characteristics** usually includes submitter's technical experience, position, and reputation. During the analysis of the email contents, we found that, if the sender is a maintainer of this component, even if the message of a complex patch is expressed in just a few words, the patch still has a chance of getting merged (ten such cases). For example, an accepted patch was adding a new feature with 123 lines of code modified, and the submitter wrote in the email: *“Add custom statistics to be reported via ethtool -S. These include driver specific per-cpu [sic] statistics as well as queue and channel counters.”* Only the information about the “what” was mentioned without any explanation about the “why”. A further exploration indicates that he is the maintainer of this module. He may have enough reputation to convince the community of the quality of his patches. As emphasized by OSS communities, trust plays a key role in facilitating their success [1].

If a patch satisfies the **reviewers' expectation** (e.g., a patch is a new feature that is really needed by the community), even if its description is inadequate, this patch may still have a chance of being accepted. We found that, if a new feature had been discussed previously, the information on “why” may be omitted (we found 20 such cases: refer to line 8 of Table 3 — Technical details). Moreover, for different types of patches, **reviewer's expectations** are also different. In our survey, a maintainer mentioned, *"I have rejected cleanup patches where the message was badly formatted. I am stricter with cleanup patches than with bug fixes."*

Through the above analysis, we found that the theme **individual factors** plays a special role in communication. Some patches without an adequate description were still accepted. However, in the survey we conducted (the details will be presented in Section 5), the developers expressed the lack of important information (e.g., motivation) due to **individual factors** was harmful to software maintenance. For CSP, the main purpose is to clearly describe the patch and convince reviewers to believe that the patch is really needed and of high quality and, therefore, it is necessary to accept. Although CSP is essentially the same as the communication in daily life (both are the act of conveying the intended meaning from one person to another), it also has differences. As described in Section 1, once the patch is accepted, the content of communication will be used as a permanent description of the submission, which is the essential media for (future) developers to understand the corresponding code fragment and has great significance to software maintenance. For example, we found that the *Email #454294* received a new reply five years after it was sent because the patch description did not provide technical details. However, the question was not answered because the developer had already left the community. Therefore, although it seems that **individual factors** may affect the descriptions of patches, it will cause serious problems in future maintenance.

4.3 Practices of Successful CSP

On the basis of the findings in the previous sections, we could identify the expression elements and the possible context of each patch. By extracting and generalizing the context in which each element appeared or did not appear, we summarized 18 preliminary practices. After performing a survey, we dropped one practice and eventually obtained 17 practices. Below, we follow the three **contextual factors** to introduce the expression elements and specific context for each of the practices, along with the examples of good and poor CSP. The practice related to the other two contextual factors **submitter's characteristics** and **reviewer's expectations** was dropped in the survey.

4.3.1 Practices Related to ***Rules of the Community****.* These practices indicate **when, which granularity, and what manner for CSP**, and they apply to all patches. Note that they may be different in other communities.

1. Choose the right timing. If the time is not ideal, the review process could be delayed, and even worse, the patch may be ignored owing to the busy work of the reviewers. In the Linux kernel community, at the beginning of each development cycle, the "merge window" is said to be open [35]. Code deemed to be sufficiently stable (which was reviewed and accepted) is merged into the mainline kernel. During that time, reviewers are pretty busy at a rate approaching 1,000 patches per day [34]. Among the new patches, only bug fixes can be reviewed. As a general rule, if a patch submitter misses the merge window for a feature, the best thing to do is to wait for the next development cycle. If the submitter does not get a reply, s/he may better wait for a minimum of one week before resubmitting and do not try to communicate with people who are preoccupied with other tasks.

2. Separate each logical change into a separate patch. This practice tells developers the level of granularity for CSP. The independence of patches facilitates a review by other kernel developers. For example, if a patch includes both bug fixes and performance enhancement, those changes should be separated into two or more patches. When submitters divide the change into a series of patches, it is important to consider the sequence of the patch set to ensure that the kernel builds and runs properly after each patch in the series. The build sequence of these patches should be clearly pointed out, e.g.,"[PATCH v2 01/27]" in the subject line means that it is the second version and the first of 27 patches in this patch set. Moreover, if the submitter makes a single change to numerous files, those changes should be grouped into a single patch.

3. When describing what you did, use the imperative mood. This practice tells developers what manner to communicate. The purpose is to make the expression more concise. For example, it is recommended to write "Fix ..." instead of "This patch ....," and "I fixed ....". The following is an example of poor CSP.

This patch allows APEI generic error source table with external IRQ to share a single IRQ. (Email #812476)

The reviewer commented," *'This patch' in a commit message is tautologically redundant.*"

4. Follow the rules, norms, and expectations in the community, such as the formatting requirement. This is one of the strictest requirements of patch submission. Developers should read each rule carefully before starting a message. We found two expression elements related to this practice (see Table 2: the elements belonging to the sub-theme — **satisfy the basic requirement of the community**). Below is an example of CSP that did not conform the rules of community.

For *Email #808912*, because it did not satisfy the formatting requirement, the reviewer commented, "*Please don't put anything below the Signed-off-by: line, you will note that all other commits are written that way.*"

4.3.2 Practices Related to ***Implementation of the Patch****.* It means that all the practices in this section are closely related to the specific implementation of patches, such as the difficulty of patches, whether there are references, and so on. These practices focus on the expression elements under the theme — **Content of CSP**, and they apply to all patches.

5. Describe the motivation convincingly, including WHY you submitted the patch and WHAT you did. The purpose of the communication is to clearly express what the patch is about and persuade the reviewers that this patch is needed. Therefore, it is important to convince the reviewers that there is a problem worth fixing or that it is a necessary feature. It makes sense for reviewers to read past the first paragraph. Below are good examples, in which the submitters clearly explained what they did and provided strong reasons for it:

Make the behavior of clk_get_rate consistent with common clk's clk_get_rate by accepting NULL clocks as parameter. Some drivers rely on it, otherwise will cause an OOPS. (Email #810737)

On v1, "tasks" and "cgroup.procs" are expected to be sorted which makes the implementation expensive and unnecessarily complicated involving result cache management. v2 doesn't have the sorting requirement, so it can just iterate and print processes one by one. seq_files are either read sequentially or reset to position zero, so the implementation doesn't even need to worry about seeking. (Email #745004)

The following practices (6–9) are related to the expression element — **description of implementation**.

6. If the implementation is complex, briefly describe the technical details. It is important to describe the change in plain English so that the reviewer can verify that the code behaves as the submitter intends to; this could save review time.

7. If there is an improvement after applying the patch, clearly describe the improvement. Use numbers to quantify the improvements in performance, memory consumption, stack footprint, or binary size, as this can obviously help reviewers to understand the value of the patch, for example:

In the following log you can see a significant difference in the code size and data segment, hence in the dec segment. This log is the output of the size command, before and after the code change: before: text:21671 data:3632 bss:128 dec:25431; after: text:2136 data:3576 bss:128 dec:25070 (Email #808446)

8. If there is a cost after applying the patch, provide the trade-off. It can help reviewers weigh the costs against the benefits to determine whether the change is worthy. For example, a patch we analyzed mentioned the following:

Add single-element dequeue functions to rcu_segcblist. It is less efficient than using the extract and insert functions, but allow more precise debugging code. (Email #745004)

Another example of poor CSP — the reviewer commented, "*This limitation should be explained in the cover letter and changelog.*" *(Email #759033)*

9. If the patch refers to other information, e.g., commits, email message, or links to related information, include this information. It can help reviewers easily understand what the patch submitter did, for example:

This bug fix patch is related to a specific commit; the submitter included the SHA-1 ID as well as a one-line summary of the commit — *Commit e21d2170f36602ae2708 ("video: remove unnecessary platform_set_drvdata()") removed the unnecessary platform_set_drvdata(), but left the variable "dev" unused, delete it. (Email #2833310)*

Reference of other's patch — *Use devm_reset_control_get_exclusive to avoid resource leakage (based on patch "Convert drivers to explicit reset API" from Philipp Zabel). (Email #812307)*

The next three practices (10–12) are related to the expression element — **description of quality**.

10. If the patch is a prototype or on-going, include the limitation of the current version. For some patches, the initial version is usually just a prototype. The code only involves the implementation of the core part. Therefore, the patch submitter needs to point out the limitations of the current patch, so that the patch will not be directly denied for these reasons:

If implemented, "max_bad_blocks" returns the maximum number of bad blocks to reserve for a MTD. An implementation for NAND is coming soon. (Email #748338)

11. If the patch has been tested or reviewed by others, use tags such as "Reviewed-by" or "Test-by." This tag serves to give credit to the reviewers and to inform the reviewers about the degree of review that has been done on the patch, which will normally increase the likelihood of acceptance.

12. If there is more than one way to implement the patch, show the superiority of the solution. OSS projects, particularly large projects such as the Linux kernel, usually have many developers who contribute. Therefore, showing the superiority of a patch can help make it stand out:

There are two ways to solve that: 1) Reserve a hotplug state for LTTNG; 2) Add a dynamic range for the prepare states. While #1 is the simplest solution, #2 is the proper one as we can convert in tree users, which do not care about ordering, to the dynamic range as well. Add a dynamic range which allows LTTNG to request states in the prepare stage. (Email #751332)

4.3.3 *Practices Related to* ***Type of the Patch****.*

Each of these practices targets a specific type of patches (e.g., a new feature or a bug fix) and emphasizes what needs to be communicated for this type. They all focus on the expression elements belonging to the theme — **Content of CSP**.

13. Provide sufficient reasons why this new feature is required, especially for the complex features. Generally, adding a new feature involves a huge amount of code, which may bring many problems and make the code difficult to maintain. If it is not very necessary, reviewers are generally reluctant to accept this type of patches. Therefore, the submitter must provide a good reason to explain why this patch is necessary. Below is an example of poor CSP that only explained what feature the patch implemented without providing any reason.

This adds a device tree definition file for LEGO MINDSTORMS EV3. (Email #728065)

This feature involves the modification of 457 lines of code, but its description is very brief. The reviewer commented,"*Can you clarify why we need it?*"

14. If the new feature includes several functions, list all of them or give a brief introduction. For the reviewers, if the functions of a patch are new, they may feel unfamiliar with them. If the code is complex, it makes the situation even worse. This practice helps the reviewers assess the importance of the patch and better prepare for the review.

15. If the patch is a bug fix, describe the scenario of the bug or how to reproduce it, or provide the link to the issue report. Providing this information can help reviewers obtain a more comprehensive understanding of what the submitter did and confirm whether s/he has solved it. The following example not only provides the scenario of the bug but also includes the possible reasons:

Markus reported that perf cannot scroll down after refresh. This was because the number of entries are [sic] not updated. (Email #725298)

16. Include the seriousness of bug as it can get more attention from reviewers. For example, reasons such as "it can lead to a system crash", or "it can not support the core device" are very convincing. Otherwise, it is difficult to understand the importance of the patch when the reviewers are inadequately aware of the scenario, which may result in the patch being rejected. The following words are usually used to describe the seriousness of the bug: *error, warning, block, crash, wrong, invalid, leak, can't support, interrupt, etc.*

17. For the patches that are considered trivial or that simplify the code, just indicate what you did. For simple cases, describing what is done reflects why it needs to be done. Unnecessary description is a waste of time for both developers and reviewers. Below is an good example:

Trivial fix to spelling mistake in dev_dbg message. (Email #10576333)

We randomly selected 200 patches that were rejected due to communication problems. By comparing the differences of CSP between these patches and their corresponding accepted versions, as well as the reviewers' comments, we analyzed the reasons why they were rejected, i.e., which practices were not followed. As shown in Fig. 2, 64 (32%) patches were rejected because of lacking the tags such as, "Tested-by" or "Reviewed-by". This can be explained because the initial versions of many patches had not been tested by multiple developers, the submitters sent them to the mailing list, and the relevant maintainers tested them (added the tags for the passed patches) to evaluate their quality. The second reason for rejection (46 (23%) patches) is failing to follow the community's rules, which indicates many developers did not know the importance of CSP and were not aware of the rules even before submitting. In addition, many patches were rejected because of lacking the descriptions of the way to reproduce the bug, motivations, and technical details. The lack of such information makes it difficult for reviewers to understand the patches. It is worth noting that some practice numbers were close to 0 mainly because the corresponding context was rare. For example, in most cases, there was only one way to implement the patch, and, therefore, it was difficult to provide the superiority of the solution.

5 VALIDATION

We validated the results by a survey with 26 responses returned. In the first part, we asked their experiences with CSP. Almost all the respondents (96%) considered communication important for patch review. Even though it has been realized that an appropriate expression of message is essential to patch acceptance, it is not easy to do it well. More than half (54%) of the respondents

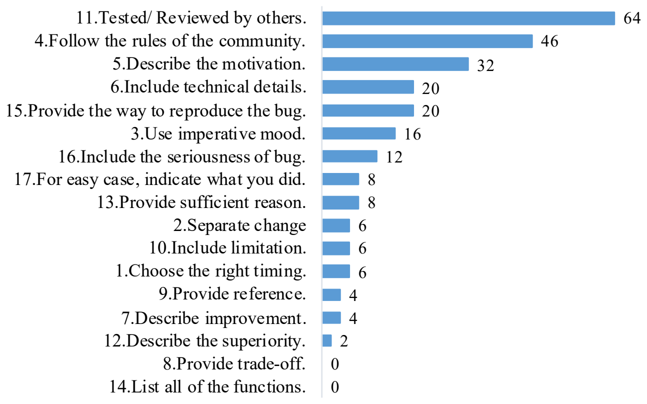


Fig. 2. Number of the Patches not Following the Specific Practices

considered it difficult to write a patch description. Worse still, substantial effort of both developers and reviewers is wasted just because the patch is not well described. Sixty-one percent of the developers had their patches rejected and 73% of the reviewers had rejected patches owing to inappropriate expression (missing or unclear description of some expression elements).

Figure 3 shows the developers' perceptions of the 16 practices, which can be interpreted as good practices. Because experts over the years have argued that the median (measure of central tendency) and frequencies (percentages of responses in each category) should be used for analyzing Likert scale data [56], we present these two types of data. In Figure 3, the importance means the extent to which the practice may help the reviewers understand the patch and therefore accept it. The operating difficulty is the amount of effort required to follow the practice or the possibility of forgetting to do so. Most of the respondents considered the 16 practices were "important" or "very important" (median >= 3), 15 of which were almost very important (median : 4 or 5). It validates our findings, indicating that the obtained practices are effective. It is interesting to notice that one of the community rules, namely, "[u]se the imperative mood," was not considered to be very important, which could be explained by a reviewer's opinion: *"[i] don't care about language seriously. [i] just need to make sure it's clear."* For all the practices, the operating difficulty was lower than the importance. Only four of them (practice #6, #8, #12, #13) were considered to be above medium in difficulty (median > 3), which suggests that almost all the practices are reasonably applicable.

Additionally, there are two candidate practices that aim to save developers time. Because they omit some information, it is difficult for us to weigh the pros and cons. Therefore, we asked the developers what they thought of them. The first one is "[i]f the feature has been discussed previously, or it's a widely recognized feature, or you (the author of the patch) have commit right, you may omit providing the reason." This is the only one practice related to **individual factors**: **reviewers' expectations** and **submitter's characteristics**. Sixty percent of the respondents thought it was meaningless and 20% thought it was not significant. They gave several explanations, e.g., *"[p]roviding this information can point people without context in the right direction"*, or *"[o]mitting this information is harmful to the code maintenance in the future."* Therefore, we removed this practice. The other practice is "[f]or the patches such as: easy case, and simplify code, just express what you did." Eighty-eight percent of the respondents considered it significant, because it can save developers' effort.

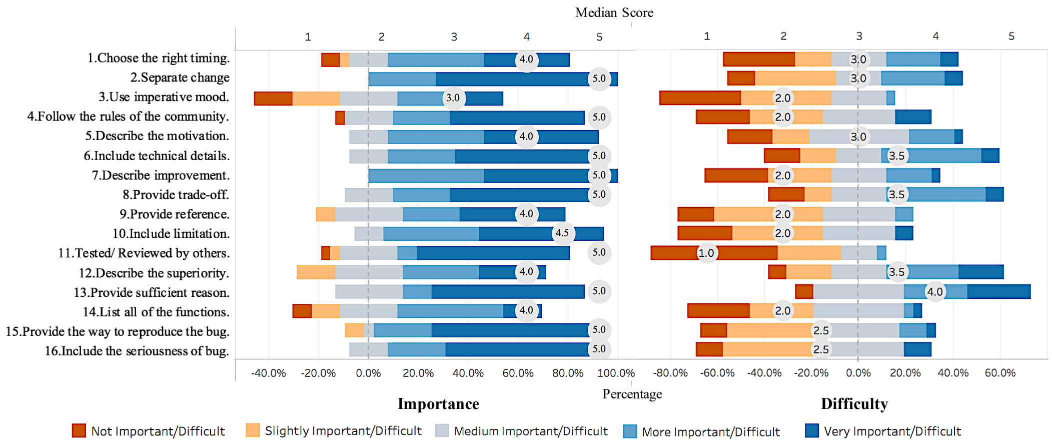


Fig. 3. Respondents' View of the Practices (Numbers on the upper horizontal axis show: 1: Not important/difficult; 2: Slightly important/difficult; 3: Medium important/difficult; 4: More important/difficult; 5: Very important/difficult.)

6 DISCUSSION

In this section, we discuss the recommendations for different practitioners including patch (or pull-request) submitters, code reviewers, and tool designers, and the implications for OSS communities and CSCW researchers.

6.1 Recommendations for Patch (or Pull-request) Submitters

Previous studies found that sending a meaningful/ correct message is difficult for developers [27] and even becomes one of the main social barriers for newcomers [53]. Through the survey with kernel developers, we also found that although developers have realized the importance of CSP, it is difficult to do it well (see Section 5). In this paper, we studied how to conduct CSP and obtained 17 practices that were considered important and actionable by developers. Among them, we highlight four practices, including three practices that are considered to be very important but difficult and one relatively difficult practice that developers often do poorly, for which we provide detailed analysis and instructions. We recommend that patch (or pull-request) submitters, especially newcomers, pay close attention to these four practices.

Three practices, including **#13. provide sufficient reasons**, **#6.include technical details**, and **#8.provide a trade-off**, are considered to be very important but difficult (see Figure 3, median of importance = 5; median of difficulty > 3). It appears that a rich and convincing description of a patch is never easy to provide. For example, if a submitter wants to give sufficient reasons, s/he has to have the acquaintance with the system, the application scenarios, and the customer requirements, etc. As for provide a trade-off, it even needs the submitter provide experimental data to compare the pros and cons (e.g., "adding this new feature doubles efficiency, but also needs 20M extra memory"). They all take developers' great effort. Since many submitters are not fully aware of the importance of doing so, the message expressions are often inadequate. Submitters should pay more attention to describing the motivation and implementation, and thus persuade reviewers that the patch is necessary and reliable. The reason can be sufficient if s/he can give evidence similar to *"[w]hat a bad effect there is without it" or "[i]t can support/enable ..."*. For complex patches, the key details of the implementation should be briefly described in plain language, which is effective for reviewers to understand it. An accurate description of the effect, including both the positive and the side effect, is essential. Otherwise, the patch might be rejected, e.g., we found a rejected patch without an explanation of the side effect and the reviewer said, *"[d]oing that allocation comes at the cost of having to do an extra pointer dereference every time you use it."*

The practice **#15. provide the way to reproduce the bug** is the reason that many bug fixing patches were rejected. Although there are more patches not obeying the practice #11 and the practice #4 (as shown in Fig. 2), #15 is relatively difficult to operate as compared to those two practices (see Fig. 3). Developers should be more careful in providing the scenario of the bug or in describing how to reproduce it, which can help reviewers to determine which problem the developer has solved, e.g., giving the execution process of the program along with error/warning information. The emphasis of these four practices is on how to attract special attention, not suggesting that other practices are unimportant.

We believe that our findings and the corresponding discussion have significant meaning for patch submitters, especially for newcomers, who are not familiar with how to describe their contribution. After all, writing perfect text information sometimes is considered more difficult than "technical" tasks like adding new features or fixing bugs [25].

6.2 Recommendations for Code Reviewers

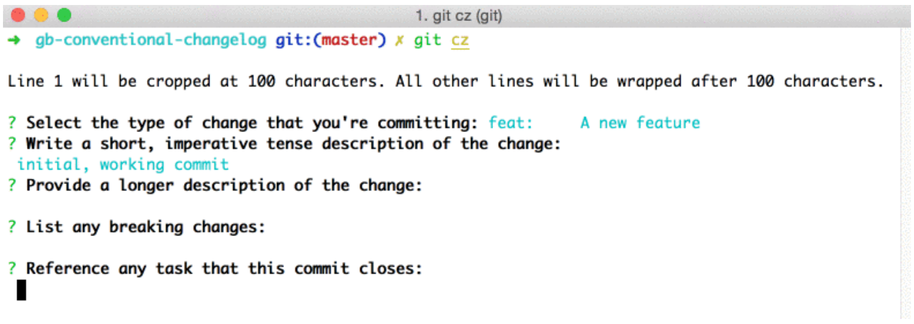
As with many human collaborative systems, software development requires effective communication among participants, i.e., developers [30]. In particular, the process from patch submission to patch review involves the collaboration between patch submitters and reviewers, in which the descriptions of code, as a communication means, plays an important role. Concise yet informative descriptions of code are effective signals in software development that reduce the information asymmetry between patch submitters and reviewers, thus speeding up the review process. As mentioned in Introduction, reviewers of large-scale OSS projects usually undertake tremendous pressure of code review. We believe that based on our findings, patch submitters can better understand how to communicate and thus, write high-quality descriptions, which can, in turn, reduce the effort required by reviewers and accelerate the process of code review.

Code review is an important process to control software projects' health, in which reviewers play a critical role as quality inspectors. However, we found that the contextual factors related to — **individual factors** may introduce potential risks to future maintenance. It means that sometimes reviewers only focus on the correctness of code but ignore the importance of code description. They do not care whether the information contained in patch descriptions is sufficient for future maintenance, as long as it can ensure effective communication with submitters at that time. Therefore, we recommend that reviewers should treat whether the description is appropriate as one of the criteria for judging whether a patch can be accepted or not. Similarly, when they submit their own code, they should also pay attention to its description.

6.3 Recommendations for Tool Designers

Some tools have been designed and applied to help developers write conventional commit messages and make them standardized in the whole team to facilitate efficient development and maintenance. For example, Git took this into account when it was designed: developers can customize their commit message by configuring "git.template" in ~/.gitconfig [13]. Once developers configure the template, Git will use that template to produce initial message when committing. Some integrated development environments also provide plug-ins to help developers write commit message, e.g., "Git Commit Template" developed by Jet Brains.[7] These tools use predefined templates to remind developers of the correct format and style when creating commit messages, but basically, they only

[7]https://plugins.jetbrains.com/plugin/9861-git-commit-template



```
1. git cz (git)
➜  gb-conventional-changelog git:(master) ✗ git cz

Line 1 will be cropped at 100 characters. All other lines will be wrapped after 100 characters.

? Select the type of change that you're committing: feat:     A new feature
? Write a short, imperative tense description of the change:
 initial, working commit
? Provide a longer description of the change:

? List any breaking changes:

? Reference any task that this commit closes:
 █
```

Fig. 4. Commitizen Template

focus on avoiding format errors and pay little attention to content. While the content is indeed difficult to fill, there is still some work that can be done. “Commitizen” is another popular interactive commit formatting tool, which were downloaded more than 50,000 times per week in 2009.[8] This tool prompts developers to fill out the required commit fields at commit time. As shown in Fig. 4, it guides developers to select the type of a patch and then complement the relevant information.

It is interesting to observe that this tool has already paid attention to the contextual factor – “type of patch” and a few implementation factors, e.g., whether referring to other information. However, many contextual factors (as discovered in this study) are still not considered. If some contextual information, such as the technical difficulty and maturity of patches, is not prompted to developers, they can easily forget to include. Based on our findings, we suggest that these tools may interactively provide contextual factors that guide developers to supplement the key elements of CSP. For example, when a developer indicates that the patch’s technical difficulty is high, the tool may remind her/him to explain the technical details in plain language. It is important to note, however, this process needs to trade off the perfection of the commit message with the effort the developer needs to devote. Therefore, it is helpful to provide a “easy case” type, which none of these tools takes into account. When developers complete such information, the tool can automatically generate standard commit messages.

From the perspective of maintainers of the projects, another drawback of these tools is that it does not enforce the team to follow this CSP style (i.e., no validation process). A better way would be to use git hooks [13] to help enforce this rule, which secures that commit messages are validated before being pushed to the server.

6.4 Implications for OSS Communities

Establishing regulations on CSP may pay back in OSS communities, especially large-scale communities. Brook’s Law claims that communication overheads increase as the number of people increases [10]. Developers have to spend more effort on keeping themselves and others know what is going on in the project, thus making them less productive. In this situation, if a community has some regulations on the format and language style of CSP, the communication burden may be partially alleviated. The Linux kernel community has spent much effort to provide a series of guidelines and rules for CSP (refer to the documents in the appendix[4]). Compared to other OSS communities, it has already done an outstanding job in this respect with years of experience. However, our study still revealed five valuable practices (practices #10, #12—15) that the Linux kernel community does not list in its guidelines. We also show that the potential risk in the descriptions

[8]https://www.npmjs.com/package/commitizen

of accepted patches can be ignored by an experienced community like the Linux kernel. These findings are of great significance to the sustainable development of the Linux kernel community. Moreover, our findings, especially the practices related to **content of CSP** (including 13 practices: practices #5—17), have great reference value for other OSS communities. Our discussion of tools also gives OSS communities a new idea to optimize CSP.

6.5 Implications for CSCW Researchers

Our preliminary investigation of CSP revealed what/how to say when submitting patches. By reviewing the related research literature within the context of CSCW, we find that although much effort has been devoted to investigate media of CMC, few studies focus on the content of CMC. Our study bridges this knowledge gap and provides a starting point to guiding developers to conduct CSP. More work is necessary on specific research topics, such as evaluating (and creating ways to measure) the application effect of these practices, and identifying the problems that developers meet when adopting these practices and providing solutions. Researchers can also adapt the practices we proposed to different projects to build new social and collaborative systems/tools to facilitate CSP. Moreover, we believe that the CSCW researchers can benefit from referring to our research framework (expression elements -> contextual factors -> practices). Future research ought to investigate the content of communication under different circumstances of software development, and eventually build models and theories about communication in open collaboration communities.

7 THREATS TO VALIDITY

7.1 Threats to Internal Validity

Distinguishing patches with effective communication from patches with ineffective communication brings threat to the validity. A code modification might be rejected first and then accepted because of the change of circumstances, e.g., change of requirements. Our approach would identify that the earlier rejection was due to a communicating problem (while it is not). In order to alleviate this threat, we restrict the time interval between these two versions to be less than six months, because the review time is usually no more than two development cycles [33]. We randomly selected 50 sets of these patches to check and the result confirmed the classification is accurate.

The extraction of the elements of communication and practices presents threat of subjectivity. To minimize this threat, two authors individually inspected the patches and finally reached agreement through a number of discussions (the discrepancies were resolved by introducing an experienced colleague). The agreement level (0.96) is relatively high, which ensures inter-rater reliability [21]. The inspection results were then confirmed and complemented by the online developer survey.

7.2 Threats to External Validity

Since we have only studied one OSS project, some findings might not generalize to other open and closed source projects, even in the same domain. Especially, the elements and practices that relate to **rules of the community** may be different for other projects. However, the Linux kernel we studied is very large and works on one of the most used code bases around, which has abundant mature experiences. It has been using mailing list for submitting and reviewing code for nearly 30 years. Its success is inseparable from its practices on communication. Therefore, most results, especially the practices related to **content of CSP**, have great reference value and guidance significance, especially for novices entering in open source development and managers of new potential software projects. These practices can also be applied to the similar communication scenarios, e.g., sending a pull request.

8 CONCLUSION

In this study, we investigated how developers communicate in the Linux kernel community when submitting patches. By analyzing the scenarios of communicating patches, we found four themes that characterize how a message is expressed: **the timing/ granularity/ content/ manner of CSP**, and three themes that outline the context of CSP: **rules of the community, patch factors, and individual factors**. By fully considering the expression and context of the communication, as well as by conducting an online survey, we summarized 17 practices. We found that the **individual factors** plays a special role in the communication and may lead to potential problems of software maintenance. Perfect communication may not only get the patch accepted, but also may provide necessary information for the future developers. Based on the findings, we highlight four practices that deserve special attention from patch submitters and recommend reviewers to avoid the potential risks led by **individual factors**. We also discuss the recommendations for designing and improving the existing commit message formatting tools and the implications for OSS communities and the CSCW researchers.

To the best of our knowledge, our work is the first to deeply study how to guide developers what/how to say when submitting patches. We expect our results to optimize CMC in software development and benefit both patch submitters and reviewers alike: patch submitters can better understand how to communicate and thus better describe the patches to avoid being rejected; patch reviewers, in turn, can benefit from the improved quality of patches and reduced review effort. We hope that the community will realize the role of **individual factors** in communication and the problems of communication even in accepted patches. Understanding these issues is beneficial for the long-term healthy development of the community. We also expect that our research framework can provide help for investigating communication in other scenarios of open collaboration communities.

9 ACKNOWLEDGEMENTS

We would like to thank Andrew Begel for his valuable feedback on the paper. We also thank the reviewers for their great suggestions to improve the paper. This work is supported by the National key R&D Program of China Grant 2018YFB10044201, the National Natural Science Foundation of China Grants 61432001, 61690200, and 61825201.

REFERENCES

[1] Maria Antikainen, Timo Aaltonen, and Jaani Väisänen. 2007. The role of trust in OSS communities-case Linux Kernel community. In *IFIP International Conference on Open Source Systems*. Springer, Germany, 223–228.

[2] Jorge Aranda and Gina Venolia. 2009. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 298–308.

[3] Saulius Astromskis, Gabriele Bavota, Andrea Janes, Barbara Russo, and Massimiliano Di Penta. 2017. Patterns of developers behaviour: A 1000-hour industrial study. *Journal of Systems and Software* 132 (2017), 85–97.

[4] Gabriela Avram, Liam Bannon, John Bowers, Anne Sheehan, and Daniel K. Sullivan. 2009. Bridging, Patching and Keeping the Work Flowing: Defect Resolution in Distributed Software Development. *Computer Supported Cooperative Work (CSCW)* 18, 5 (16 Sep 2009), 477.

[5] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. 2009. *Communication, collaboration, and bugs: The social nature of issue tracking in software engineering.* Technical Report. University of Calgary.

[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 308–318.

[7] Nathan Bos, Judy Olson, Darren Gergle, Gary Olson, and Zach Wright. 2002. Effects of Four Computer-mediated Communications Channels on Trust Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '02)*. ACM, New York, NY, USA, 135–140. https://doi.org/10.1145/503376.503401

[8] John Waite Bowers, Donald C. Bryant, Richard W. Budd, Robert K. Thorp, and Lewis Donohew. 1968. Content Analysis of Communications. *College Composition & Communication* 19, 1 (1968), 53.

[9] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work (CSCW '10)*. ACM, New York, NY, USA, 301–310.

[10] Frederick P. Brooks, Jr. 1995. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[11] Gerardo Canfora, Massimiliano Di Penta, Stefano Giannantonio, Rocco Oliveto, and Sebastiano Panichella. 2013. YODA: Young and Newcomer Developer Assistant. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1331–1334.

[12] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW '06)*. ACM, New York, NY, USA, 353–362.

[13] Scott Chacon and Ben Straub. 2014. *Pro Git* (2nd ed.). Apress, Berkely, CA, USA.

[14] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 396–407.

[15] Henrik Munkebo Christiansen. 2007. Meeting the Challenge of Communication in Offshore Software Development. In *Software Engineering Approaches for Offshore and Outsourced Development*, Bertrand Meyer and Mathai Joseph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–26.

[16] PostgreSQL community. 2013. Submitting a Patch. Retrieved January, 2019 from https://wiki.postgresql.org/wiki/Submitting_a_Patch

[17] J. Corbet, G. Kroah-Hartman, and A. McPherson. 2012. Linux Kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. Retrieved January, 2019 from https://go.linuxfoundation.org/who-writes-linux-2012

[18] Alan Cox, Alexander Viro, and et al. 2015. The linux-kernel mailing list FAQ. Retrieved January, 2019 from http://vger.kernel.org/lkml/

[19] Daniela S Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, USA, 275–284. https://doi.org/10.1109/ESEM.2011.36

[20] Sarah D'Angelo and Andrew Begel. 2017. Improving Communication Between Pair Programmers Using Shared Gaze Awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6245–6290.

[21] Jason Davey, P. Gugiu, and Chris Coryn. 2010. Quantitative Methods for Estimating the Reliability of Qualitative Data. *Journal of MultiDisciplinary Evaluation* 6, 13 (2010), 140–162. http://journals.sfu.ca/jmde/index.php/jmde_1/article/view/266

[22] Cleidson de Souza, Jon Froehlich, and Paul Dourish. 2005. Seeking the Source: Software Source Code As a Social and Technical Artifact. In *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work (GROUP '05)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1099203.1099239

[23] Center for Literacy Studies of the University of Tennessee. 2011. Communication Process. Retrieved January, 2019 from http://www.cls.utk.edu/pdf/ls/Week1_Lesson7.pdf

[24] Adrian Furnham. 1982. The message, the context and the medium. *Language & Communication* 2, 1 (1982), 33–47.

[25] R. Stuart Geiger, Nelle Varoquaux, Charlotte Mazel-Cabasse, and Chris Holdgraf. 2018. The Types, Roles, and Practices of Documentation in Data Analytics Open Source Software Libraries. *Computer Supported Cooperative Work (CSCW)* 27, 3 (01 Dec 2018), 767–802.

[26] Barney G Glaser. 1965. The Constant Comparative Method of Qualitative Analysis. *Social Problems* 12, 4 (1965), 436–445.

[27] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: the contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, IEEE, USA, 285–296.

[28] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, IEEE, USA, 358–368.

[29] Carl Gutwin, Reagan Penner, and Kevin Schneider. 2004. Group Awareness in Distributed Software Development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW '04)*. ACM, New York, NY, USA, 72–81.

[30] James D Herbsleb. 2007. Global software engineering: The future of socio-technical coordination. In *Future of Software Engineering (FOSE'07)*. IEEE, IEEE, Minneapolis, MN, USA, 188–198.

[31] James D. Herbsleb, David L. Atkins, David G. Boyer, Mark Handel, and Thomas A. Finholt. 2002. Introducing Instant Messaging and Chat in the Workplace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '02)*. ACM, New York, NY, USA, 171–178. https://doi.org/10.1145/503376.503408

[32] A Hindle, D. M German, M. W Godfrey, and R. C Holt. 2012. Automatic classication of large changes into maintenance categories. In *IEEE International Conference on Program Comprehension*. IEEE, USA, 30–39.

[33] Y. Jiang, B. Adams, and D. M. German. 2013. Will my patch make it? And how fast? Case study on the Linux kernel. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, USA, 101–110. https://doi.org/10.1109/MSR.2013.6624016

[34] The kernel development community. 2017. How the development process works. Retrieved January, 2019 from https://www.kernel.org/doc/html/v4.11/_sources/process/2.Process.rst.txt

[35] The kernel development community. 2017. Submitting patches: the essential guide to getting your code into the kernel. Retrieved January, 2019 from https://www.kernel.org/doc/html/v4.11/_sources/process/submitting-patches.rst.txt

[36] Andrew J Ko, Brad A Myers, and Duen Horng Chau. 2006. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, IEEE, USA, 127–134.

[37] Ben Linders. 2017. Making Distributed Development Work. Retrieved January, 2019 from https://www.infoq.com/news/2017/03/distributed-development

[38] Björn Lundell, Brian Lings, Pär J. Ågerfalk, and Brian Fitzgerald. 2006. The distributed open source software development model: observations on communication, coordination and control. In *Proceedings of the Fourteenth European Conference on Information Systems, ECIS 2006, Göteborg, Sweden, 2006*. IEEE, USA, 683–694. http://aisel.aisnet.org/ecis2006/41

[39] Ray Madachy. 2007. Distributed Global Development Parametric Cost Modeling. In *Software Process Dynamics and Agility*, Qing Wang, Dietmar Pfahl, and David M. Raffo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–168.

[40] Mark Mason. 2010. Sample Size and Saturation in PhD Studies Using Qualitative Interviews. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* 11 (08 2010).

[41] David W. McDonald and Mark S. Ackerman. 2000. Expertise Recommender: A Flexible Recommendation System and Architecture. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00)*. ACM, New York, NY, USA, 231–240. https://doi.org/10.1145/358916.358994

[42] Tom Mens, Maálick Claes, Philippe Grosjean, and Alexander Serebrenik. 2014. *Studying Evolving Software Ecosystems based on Ecological Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 297–326. https://doi.org/10.1007/978-3-642-45398-4_10

[43] A. Mockus and J. Herbsleb. 2001. Challenges of Global Software Development. In *Proceedings of the 7th International Symposium on Software Metrics (METRICS '01)*. IEEE Computer Society, Washington, DC, USA, 182–. http://dl.acm.org/citation.cfm?id=823456.824019

[44] Audris Mockus and James D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *International Conference on Software Engineering*. IEEE, USA, 503–512.

[45] Kumiyo Nakakoji, Yunwen Ye, and Yasuhiro Yamamoto. 2010. *Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments*. Springer Berlin Heidelberg, Berlin, Heidelberg, 219–236. https://doi.org/10.1007/978-3-642-10294-3_11

[46] Tuomas Niinimaki. 2011. Face-to-Face, Email and Instant Messaging in Distributed Agile Software Development Project. In *Sixth IEEE International Conference on Global Software Engineering Workshop*. IEEE, USA, 78–84.

[47] Peter C. Rigby and Margaret Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *International Conference on Software Engineering*. IEEE, USA, 541–550.

[48] R Rivest. 1992. The MD5 Message-Digest Algorithm. *Rfc* 473, 10 (1992), 492–492.

[49] K. Schmidt. 1994. Cooperative work and its articulation: requirements for computer support. *Le Travail Humain* 57, 4 (1994), 345–366.

[50] Kjeld Schmidt and Carla Simone. 1999. Coordination mechanisms: towards a conceptual foundation of CSCW systems design. Computer Support Coop Work. J Collaborative Comput 5(2/3):155-200. *Computer Supported Cooperative Work* 5 (11 1999).

[51] Edward Smith, Robert Loftin, Emerson Murphyhill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, San Francisco, CA, USA, 89–92.

[52] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. 2015. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM conference on Computer supported cooperative work (CSCW) &amp; social computing*. ACM, ACM, New York, NY, USA, 1379–1392.

[53] Igor Steinmacher, Marco Gerosa, Tayana U Conte, and David F Redmiles. 2019. Overcoming Social Barriers When Contributing to Open Source Software Projects. *Computer Supported Cooperative Work (CSCW)* 28, 1-2 (2019), 247–290.

[54] Margaret-Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M German. 2016. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering* 43, 2 (2016), 185–204.

[55] Margaret Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M. German. 2017. How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. *IEEE Transactions on Software Engineering* 43, 2 (2017), 185–204.

[56] Gail M Sullivan and Anthony R Artino Jr. 2013. Analyzing and interpreting data from Likert-type scales. *Journal of graduate medical education* 5, 4 (2013), 541–542.

[57] Yida Tao, Donggyun Han, and Sunghun Kim. 2014. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE Computer Society, Washington, DC, USA, 271–280.

[58] Nelson Nunes Tenório Junior, Danieli Pinto, and Pernille Bjørn. 2018. Accountability in Brazilian Governmental Software Project: How Chat Technology Enables Social Translucence in Bug Report Activities. *Computer Supported Cooperative Work (CSCW)* 27 (06 2018), 715–740.

[59] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. 2017. Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ Computer Science* 3 (2017), e111.

[60] M. Rita Thissen, Jean M. Page, Madhavi C. Bharathi, and Toyia L. Austin. 2007. Communication Tools for Distributed Software Development Teams. In *Proceedings of the 2007 ACM SIGMIS CPR Conference on Computer Personnel Research: The Global Information Technology Workforce (SIGMIS CPR '07)*. ACM, New York, NY, USA, 28–35. https://doi.org/10.1145/1235000.1235007

[61] Linus Torvalds and David Diamond. 2001. Just for Fun: The Story of an Accidental Revolutionary. *Harperbusiness* 238, 6-7 (2001), S87.

[62] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's Talk About It: Evaluating Contributions Through Discussion in GitHub. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 144–154. https://doi.org/10.1145/2635868.2635882

[63] Pei-Yun Tu, Chien Wen (Tina) Yuan, and Hao-Chuan Wang. 2018. Do You Think What I Think: Perceptions of Delayed Instant Messages in Computer-Mediated Communication of Romantic Relations. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 101, 11 pages. https://doi.org/10.1145/3173574.3173675

[64] Yulin Xu and Minghui Zhou. 2018. A Multi-level Dataset of Linux Kernel Patchwork. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 54–57. https://doi.org/10.1145/3196398.3196475

[65] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for it: determinants of pull request evaluation latency on GitHub. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, IEEE, USA, 367–371.

[66] Nabi Zamani. 2018. Generating release notes from git commit messages using basic shell commands (git/grep). Retrieved June, 2019 from https://blogs.sap.com/2018/06/22/generating-release-notes-from-git-commit-messages-using-basic-shell-commands-gitgrep/

[67] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. 2017. On the Scalability of Linux Kernel Maintainers' Work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 27–37.

[68] Minghui Zhou and Audris Mockus. 2012. What Make Long Term Contributors: Willingness and Opportunity in OSS Community. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 518–528. http://dl.acm.org/citation.cfm?id=2337223.2337284

[69] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2016. Effectiveness of Code Contribution: From Patch-based to Pull-request-based Tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 871–882.

Received April 2019; revised June 2019; accepted August 2019