Suboptimal Comments in Java Projects: From Independent Comment Changes to Commenting Practices

CHAO WANG and HAO HE, School of Computer Science, Peking University and Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
UMA PAL, College of Information and Computer Sciences, University of Massachusetts Amherst, USA
DARKO MARINOV, Department of Computer Science, University of Illinois at Urbana-Champaign, USA
MINGHUI ZHOU, School of Computer Science, Peking University and Key Laboratory of High Confidence Software Technologies, Ministry of Education, China

High-quality source code comments are valuable for software development and maintenance, however, code often contains low-quality comments or lacks them altogether. We name such source code comments as suboptimal comments. Such suboptimal comments create challenges in code comprehension and maintenance. Despite substantial research on low-quality source code comments, empirical knowledge about commenting practices that produce suboptimal comments and reasons that lead to suboptimal comments are lacking. We help bridge this knowledge gap by investigating (1) *independent comment changes (ICCs)*—comment changes committed independently of code changes—which likely address suboptimal comments, (2) commenting guidelines, and (3) comment-checking tools and comment-generating tools, which are often employed to help commenting practice—especially to prevent suboptimal comments.

We collect 24M+ comment changes from 4,392 open-source GitHub Java repositories and find that ICCs widely exist. The *ICC ratio*—proportion of ICCs among all comment changes—is ~15.5%, with 98.7% of the repositories having ICC. Our thematic analysis of 3,533 randomly sampled ICCs provides a three-dimensional taxonomy for *what* is changed (four comment categories and 13 subcategories), *how* it changed (six commenting activity categories), and *what factors* are associated with the change (three factors). We investigate 600 repositories to understand the prevalence, content, impact, and violations of commenting guidelines. We find that only 15.5% of the 600 sampled repositories have any commenting guidelines. We provide the first taxonomy for elements in commenting guidelines: where and what to comment are particularly important. The repositories without such guidelines have a statistically significantly higher ICC ratio, indicating the negative impact of the lack of commenting guidelines. However, commenting guidelines are not strictly followed: 85.5% of checked repositories have violations. We also systematically study how developers use two kinds of tools, comment-checking tools and comment-generating tools, in the 4,392 repositories. We find that the use of `Javadoc` tool is negatively correlated with the ICC ratio, while the use of `Checkstyle` has no statistically significant correlation; the use of comment-generating tools leads to a higher ICC ratio.

We acknowledge the support of the National Key R&D Program of China Grant 2018YFB1004201, the National Natural Science Foundation of China Grants 61825201, and the US National Science Foundation grant IIS-2016908.
Authors' addresses: C. Wang, H. He, and M. Zhou (corresponding author), School of Computer Science, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, 100871, China and Key Laboratory of High Confidence Software Technologies, Ministry of Education, China; emails: {wchao, heh, zhmh}@pku.edu.cn; U. Pal, College of Information and Computer Sciences, University of Massachusetts Amherst, 181 Presidents Drive Amherst, Amherst, MA, 01003, United States; email: upal@umass.edu; D. Marinov, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave, Champaign, IL, 61820, United States; email: marinov@illinois.edu.

1049-331X/2023/03-ART45 $15.00
https://doi.org/10.1145/3546949

To conclude, we reveal issues and challenges in current commenting practice, which help understand how suboptimal comments are introduced. We propose potential research directions on comment location prediction, comment generation, and comment quality assessment; suggest how developers can formulate commenting guidelines and enforce rules with tools; and recommend how to enhance current comment-checking and comment-generating tools.

CCS Concepts: • **Software and its engineering → Documentation**; **Maintaining software;**

Additional Key Words and Phrases: Code comments, software documentation, coding guidelines, software evolution

ACM Reference format:
Chao Wang, Hao He, Uma Pal, Darko Marinov, and Minghui Zhou. 2023. Suboptimal Comments in Java Projects: From Independent Comment Changes to Commenting Practices. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 45 (March 2023), 33 pages.
https://doi.org/10.1145/3546949

1 INTRODUCTION

Software developers frequently write comments[1] along with source code. Comments are considered an essential form of documentation [35], are present in almost all software systems [4, 52], provide valuable information for program comprehension [4, 52], and are known to especially help developers to understand code written by others [76]. For example, Google's coding style guide states "comments are absolutely vital to keep our code readable" [60].

However, comments do not directly impact software functionality, thus many comments may not have sufficient quality, may not be properly maintained, or may be missing altogether. We use the term *suboptimal comments* to refer to all such cases of comments with insufficient quality (inconsistent, obsolete, useless, missing important information, etc.) or even missing (where comments would be desirable but are not present). Such suboptimal comments create challenges in code maintenance and reuse [102, 109], but are still prevalent in practice. For example, simple searches with "missing Javadoc" and "outdated comment" of GitHub (in August 2021) return 24K+ and 47K+ issues, respectively.

Despite substantial research on low-quality comments [10, 56, 58, 66, 69, 75, 81, 82, 86, 91, 98, 99, 102–104, 119], with the majority of prior work focused on developing automated tools for finding low-quality comments, empirical knowledge remains sparse about commenting practices that produce suboptimal comments and reasons that lead to suboptimal comments (i.e., how suboptimal comments are introduced). Bridging this knowledge gap can help researchers to focus their efforts and help practitioners to improve comment quality, follow best practices for commenting, and build better tools to help commenting.

We therefore study *independent comment changes (ICCs)*—comment changes committed independently of code changes—which likely address suboptimal comments, to attain this knowledge. We propose to study ICCs because directly identifying suboptimal comments is still *challenging* despite recent research progress: there is no unified metric for comment quality [97], and matching code and comments is difficult in general [11, 80, 112], especially for non-Javadoc comments. In contrast, ICCs can be automatically identified, and likely address suboptimal comments, by adding new comments, deleting poor comments, or updating existing comments. ICCs do not cover all suboptimal comments, but they are much more likely to indicate a suboptimal comment than non-ICCs that modify both comment and code together. We ask the following research question to explore

[1]Throughout this paper, the word "comments" refers to source code comments, *not* other types of comments, e.g., comments in an issue report.

whether ICCs can be a proxy for studying suboptimal comments: **RQ1:** ***Do ICCs target suboptimal comments; how prevalent are ICCs?*** We then study ICCs to investigate what, how, and why suboptimal comments are produced: **RQ2:** ***What comments are changed independently and how? What factors are associated with the changes?***

On the other hand, a variety of mechanisms are employed to help commenting practices and prevent suboptimal comments. Commenting guidelines, comment-checking tools, and comment-generating tools are such mechanisms that have been commonly adopted in software projects [62, 87, 108]. Commenting guidelines intuitively play an essential role in guiding consistent practices in a project [87], especially when developers come from diverse backgrounds with different opinions such as in an open-source software project [106]. Comment-checking and comment-generating tools are naturally used by developers to reduce the commenting work. However, to what extent they are used, whether they are effective, and whether they are introducing instead of preventing suboptimal comments have not been studied. We therefore ask the following research questions: **RQ3:** ***How prevalent are commenting guidelines? What guidelines are specified; are they violated?*** **RQ4:** ***How prevalent is the use of tools to assist commenting practice? Are the tools effective?***

To answer the questions, we collect 24M+ comment changes (12M+ Javadoc, 12M+ non-Javadoc) from commits of 4,392 GitHub open-source Java repositories. We apply heuristics to identify ICCs from collected comment changes, which are shown to be effective with high precision (96.4%) and recall (90.0%). Our manual inspection of randomly sampled 3,600 ICCs confirms that almost all comment changes retrieved with our method (98.14%, 3,533/3,600) are indeed ICCs, and the vast majority (92.87%, 3,281/3,533) of the ICCs target suboptimal comments. We find that ICCs exist widely: 98.7% (4,334/4,392) of the repositories have some ICC, and ∼15.5% of all 24M+ comment changes are ICCs. To evaluate the quality of commenting practice, we introduce a metric: *ICC ratio*, i.e., the proportion of ICCs among a set of comment changes. A higher ICC ratio is likely to indicate lower quality of the certain aspect of commenting practice, i.e., developers do not modify comments timely. We find that the ICC ratio tends to slightly decrease as repositories mature.

We conduct a thematic analysis of above sampled 3,533 ICCs. For each ICC, we identify *what* is changed and *how*; for 400 ICCs that have sufficient information in the commit message, we also identify *associated factors* that explain the changes. We create a three-dimensional taxonomy that explains *what* (with four comment categories, thirteen subcategories), *how* (with six commenting activity categories), and associated factors (with three high-level factors). The three factors are: *Divergence* (including all ICCs that change comments to follow a specific convention), *Tool* (including all ICCs with commit messages that explicitly mention the use of tools), and *Procrastination* (including ICCs with outdated or duplicate comments that should have been managed earlier).

We randomly sample 600 repositories to check whether they have commenting guidelines and which elements are in those guidelines. We find that commenting guidelines are not prevalent in repositories: only 15.5% (93/600) of repositories have any commenting guidelines. We establish the first taxonomy for elements in commenting guidelines, finding that they often address *where* comments should (not) be written and *what* should (not) be written in the comments. We find that repositories with such guidelines have a statistically significantly lower ICC ratio, indicating that these elements in the guidelines may improve commenting practice. However, repositories with guidelines still have violations: we find violations in 66.7% (8/12) of automatically testable subcategories and in 85.5% (59/69) repositories. We open GitHub issues for all 24 actively maintained repositories out of 59 repositories with violations. Most respondents (11/13) confirm their desire to correct the violations.

We systematically study the use of comment-checking and comment-generating tools via their configurations, related ICCs, and a survey of active developers. For comment-checking tools, we

find that 59% (2,599/4,392) of projects incorporate `Javadoc`[2] in their build files and 28% (1,253/4,392) incorporate `Checkstyle`, but the use of the tools is imperfect: developers often overlook warnings raised by these tools and tend to fix warnings in batch, which is validated by our survey. The paired one-tailed t-test suggests that repositories have a significantly lower ICC ratio after the use of `Javadoc`, but not after the use of `Checkstyle`. For comment-generating tools, we find that current comment-generating tools generate only templates or uninformative comments, which are often updated manually according to the survey. Many Javadoc skeletons are deleted (19.1% of all 201,518 deleted Javadoc ICCs) but exist for a long time (median 431 days) before being deleted. A considerable number of skeletons remain in code, accounting for ~2.5% of all Javadocs. Our survey respondents demonstrate a negative attitude toward current comment-generating tools.

In summary, this paper makes the following contributions:

- **New Method:** We propose ICCs, and an automatic technique to identify ICCs, as a novel proxy to study suboptimal comments.
- **Taxonomy:** We provide a three-dimensional taxonomy for ICCs: what is changed, how it changed, and what factor is associated with the change. We also provide the first taxonomy for elements of commenting guidelines: where and what to comment are particularly important.
- **Commenting Practice Findings:** We obtain several findings related to the prevalence, practice, and impact (on ICC ratio) of commenting guidelines and comment-checking tools, which inspire recommendations for research, practices, and tools.
- **Dataset:** We provide a dataset of comment changes and ICCs from 4,392 open-source Java repositories, our code books and taxonomies for inspected ICCs and commenting guidelines, and scripts to identify comment-checking tools [100].

2 SELECTION OF REPOSITORIES

We select repositories from GitHub, considering several criteria. Following prior studies [41, 42, 112], we only target Java projects because of Java's maturity and popularity [46, 47]. We select repositories with mature practices (with considerable stars, forks, and contributors), and rich development history (even if the repositories meanwhile became inactive), so that we can observe representative and sufficient commenting practice, and obtain lessons that could broadly generalize to even more than these open-source projects.

We use Libraries.io [73] to obtain the GitHub repository metadata. To exclude personal or toy repositories, we only keep repositories with at least ten stars, ten forks, and five contributors, as in a prior study [53, 112]. We also exclude forked repositories and obtain 8,252 repositories after this step. We further select repositories with more than 500 commits (following prior studies [51, 112]) to obtain sufficient history that exhibits comment evolution. We obtain 4,465 repositories after this step. We then exclude repositories that may not be real software projects, matching keywords "guide", "tutorial", "pattern", "note", "code", or "interview". We also exclude repositories that we cannot clone because they are removed or turned into private (e.g., `gncloud/fastcatsearch` was removed from GitHub), and finally obtain 4,392 repositories.

3 RQ1: DO ICCS TARGET SUBOPTIMAL COMMENTS; HOW PREVALENT ARE ICCS?

3.1 Methodology

Automatically Identifying Likely ICCs. To identify ICCs, we first collect hunks with comment-line changes. After cloning the 4,392 repositories, we use `git` to retrieve all commits

[2]We use different fonts to distinguish whether the word "*Javadoc*" indicates the `Javadoc` tool or Javadoc comments.

Table 1. Statistics of Comment Changes in Selected Repositories

#Repositories	4,392
#Commits	28,020,418
#Hunks with a comment-line change	25,971,427
#Hunks with only comment-line changes	9,642,818 (37.1%)

in each repository. Each commit may have one or several file changes, and we retrieve all changes of Java files. Each file change was split into one or more *hunks* by Git with some complex heuristic algorithms to make sure that long context would be folded and nearby related changed lines would be in the same hunk; e.g., to measure the "*badness*" of splitting, the "*split_score*" is calculated with heuristics based on relevant characteristics such as the number of blank lines [21]. Each hunk may contain multiple added or deleted lines. For each such changed line, we determine whether it is a *comment-line*—i.e., it contains a comment (fragment) or is a part of a comment—using three simple but accurate heuristics. We first use one regular expression to match an entire hunk to see if it contains a multi-line comment; if so, we label all the lines that match the regex as comment-lines. We then use a regex to match each changed line to see if it contains a non-Javadoc comment. Next, if some changed lines start with a star, we use a regex to see if an incomplete Javadoc exists and label related lines. Table 1 shows some statistics of our dataset.

We next decide which hunks are ICCs. Deciding whether a comment change has a corresponding code change in the same commit is an open challenge with no general solution [112]. We employ two simple strategies for the two types of comments, Javadoc and non-Javadoc comments. For Javadoc, we consider only comments for non-abstract methods and classes (including interfaces and abstract classes) because we can precisely find their corresponding code. The corresponding code of a Javadoc is source code in the method or class it describes. For each changed Javadoc comment, we analyze the entire file and check if any code line in the corresponding method or class changed. If not, we consider this comment change as a Javadoc ICC. For non-Javadoc comments, we first exclude license headers. Based on the finding that about 90% of comments are about nearby code [50], we assume that code changes in the same hunk correspond to non-Javadoc comments, as `git` applies complex heuristics to ensure one hunk contains *all* nearby changes [45]. Thus, for non-Javadoc comments, we determine whether all changed lines in a hunk are comment-lines or not. If yes, we consider the entire hunk as *one* non-Javadoc ICC, even if multiple comment-lines were changed. In general, it would be hard to count semantically separate comments that were changed when a hunk has multiple comment-lines changed.

To evaluate the effectiveness of our heuristics, we randomly sample 400 comment changes from all 24M+ collected comment changes, ensuring a confidence level of 95% and a confidence interval of 5%, following prior work [43, 51, 84, 105, 117]. Two authors manually read through the whole commit (including the commit message and all code changes) and discuss together to determine if the sampled comment change is ICC, i.e., if the comment is changed without corresponding code changes. All conflicts are resolved by a non-author arbitrator with more than six years of Java experience. We then introduce random guessing classification algorithm [68] as baseline to verify the effect of our heuristic approach. We apply our heuristics and random guessing on sampled comment changes. For each comment change, it has a 22.5% probability of being classified as ICC by random guessing, matching the ratio of ICCs we manually identified within 400 sampled comment changes. To avoid errors caused by the randomness, we conduct ten repetitions of the experiment for the random guess and calculate the average for each metric.

Inspecting Identified Likely ICCs. Our key insight is that studying ICCs is a novel way to learn about suboptimal comments, and we therefore inspect two points: (1) whether the comment

changes that our strategies identify as ICCs are indeed independent of code changes in the same commit; and (2) whether the ICCs target suboptimal comments. We sample 3,600 ICCs and conduct manual inspections. We distinguish six types of ICCs, i.e., added/deleted/updated for Javadoc/non-Javadoc ICCs, and randomly sample 600 ICCs of each type (95% confidence level, 4% confidence interval [101]).[3]

For the first point, we inspect the whole commit to check whether the comment change is triggered by any other code change in the same commit. If not, we consider it independent. The manual inspection is conducted and discussed by two authors together. All conflicts are resolved by the non-author arbitrator mentioned above. Two authors read through the whole commit (including the commit message and all code changes) and determine if the comment change is associated with other changes, e.g., the comment is changed due to the rename of methods in other files.

For the second point, following the same procedure explained above, we manually inspect the whole commit to check whether the comment change makes improvement in any way. Based on the documentation quality attributes proposed by Zhi et al. [118] that are applicable to comments, we merge attributes with similar meaning and obtain three criteria used in the inspection.

- Information–whether changed comment is more informative, corresponding to the *Completeness* [118].
- Accuracy–whether changed comment is accurate in describing related code, corresponding to the *Accuracy* and *Correctness* [118].
- Readability–whether changed comment is easier to understand via reformatting, fixing typos, translation, or adopting consistent terms, corresponding to the *Consistency*, *Format*, *Readability*, and *Spelling and grammar* [118].

If an ICC satisfies any of the above criteria, we consider it indeed targets a suboptimal comment.

3.2 Results

Table 2 presents the performance of our heuristics and the random guessing on the sampled 400 comment changes, indicating that **our heuristics can effectively identify ICCs**. The precision of our heuristics is 96.4%; the false positive ICCs are associated with faraway code changes. The recall of our heuristics is 90.0%; we may miss ICCs with nearby irrelevant code changes.

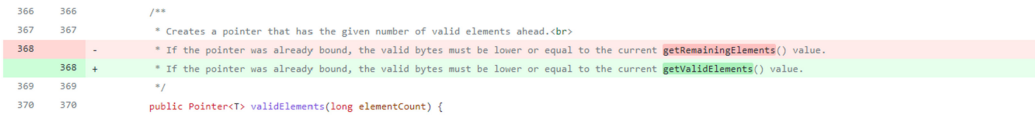
We find that **the ICCs we automatically retrieved are generally independent of code changes in the same commit**. According to our manual inspection on sampled 3,600 ICCs, 399 (11.1%) of ICCs are in commits that only change comments, so these ICCs are definitely independent as the commits have no code change at all, while others are in commits that have some code changes. We find that only 1.8% (66/3,600) of ICCs are not independent (due to renaming refactorings, changes in other files, and completed TODOs). For example, as shown in Figure 1, the comment is changed due to the renaming of involved API, where the comment is in sync with the code both before and after the commit. We also find one false positive with no comment change. In the follow-up analysis, we exclude these 66+1 cases and focus on 3,533 real ICCs.

We investigate whether the 3,533 ICCs target suboptimal comments and *aim to improve overall comment quality* (although they may or may not succeed in improving quality). We find that **almost all ICCs indeed aim to improve the information, accuracy, and readability of comments**. Some ICCs add extra content and references to make comments more informative. For example, as shown in Figure 2(a), one ICC from the commit `9a8435f9` [20] in `apache/commons-math`

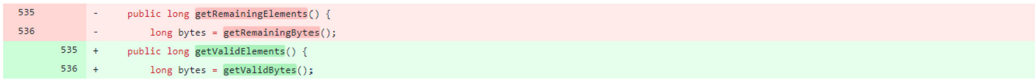
[3]An *added comment change* refers to a change that only contains new added line(s), a *deleted comment change* refers to a change that only contains deleted line(s), and an *updated comment change* refers to a change that contains both added and deleted line(s).

Table 2. The Performance of Our Heuristics and Random Guessing on 400 Sampled Comment Changes

	Precision	Recall	Accuracy	F1
Our heuristics	96.4%	90.0%	97.0%	93.1%
Random guessing	22.1%	20.4%	65.7%	21.2%



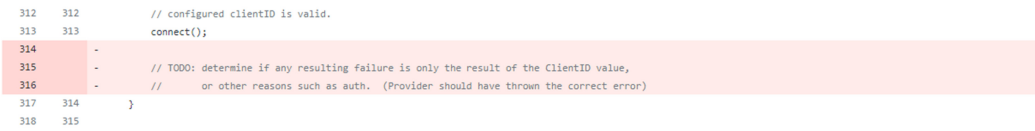
(a) The comment change



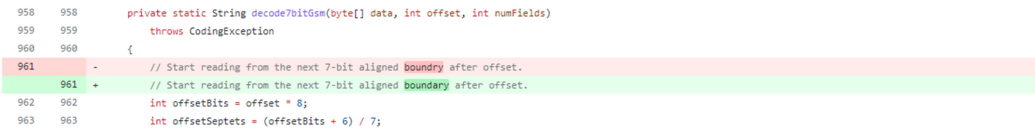
(b) Corresponding code change in the same commit

Fig. 1. An example of comment changes that are not independent (Commit `4cbff58c` in `nativelibs4java/JavaCL`).

(a) Commit `9a8435f9` in `apache/commons-math`



(b) Commit `eeb59e84` in `apache/qpid-jms`



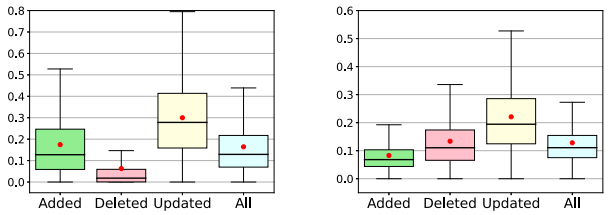
(c) Commit `bcd57322` in `aosp-mirror/platform_frameworks_base`

Fig. 2. Example ICCs from three open-source projects.

extends Javadoc to specify which methods modify instance data. Some ICCs fix or remove outdated comments to make comments more precise on explaining related code, or remove redundant comments to make information more effectively captured by readers. For example, as shown in Figure 2(b), one ICC from the commit `eeb59e84` [27] in `apache/qpid-jms` removes the TODO comment that has been addressed to reduce confusion it may bring to maintainers. Some ICCs improve the format or wording to make comments easier to understand. For example, as shown in Figure 2(c), one ICC from the commit `bcd57322` [17] in `aosp-mirror/platform_frameworks_base` fixes the typo to eliminate potential confusion. We find 252 ICCs irrelevant to improving comment quality—132 ICCs change comments that only serve tools (e.g., comments that disable code analysis in IDE), 118

Table 3. Statistics of ICCs Among All Comment Changes for Both Types of Comments and Various Changes

Change Types	Comment Type	All	Independent
Added	#Javadoc	6,639,986	1,074,784 (16.2%)
	#non-Javadoc	5,434,176	435,339 (8.0%)
Deleted	#Javadoc	4,181,405	201,518 (4.8%)
	#non-Javadoc	3,134,774	466,527 (14.9%)
Updated	#Javadoc	1,856,746	668,834 (36.0%)
	#non-Javadoc	3,504,505	981,915 (28.0%)
All	#Javadoc	12,678,137	1,945,136 (**15.3%**)
	#non-Javadoc	12,073,455	1,883,781 (**15.6%**)



(a) Javadoc ICCs (b) non-Javadoc ICCs

Fig. 3. Distributions of ICC ratios in 4,392 repositories.

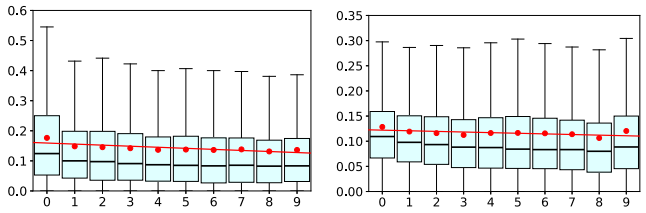
ICCs add commented code, and two ICCs we could not understand—all other 92.9% (3,281/3,533) of ICCs do aim to improve overall comment quality.

Finding 1: ICCs are likely to indicate the change of suboptimal comments: 92.9% of sampled ICCs do aim to improve overall comment quality, demonstrating the feasibility of investigating suboptimal comments via ICCs.

We find that **15.5% of all comment changes are ICCs**. Table 3 summarizes the results of our analysis of 4,392 repositories. This high ratio of ICCs suggests that a considerable number of comment changes are committed independently, where comments were likely suboptimal before the changes. The overall ICC ratios for Javadoc/non-Javadoc changes are similar (15.3% and 15.6%). For both Javadoc and non-Javadoc comments, comment updates are more likely to be ICCs, but Javadoc comments are more likely to be independently added than independently deleted, while non-Javadoc comments are more likely to be independently deleted than independently added.

We find that **ICCs exist widely in almost all repositories**. Figure 3 shows the distribution of the ratio of ICCs in each repository. Across all 4,392 repositories, the median ratios are about 12.7%/6.8%, 1.8%/11.0%, and 27.8%/19.4% for Javadoc/non-Javadoc ICCs that are added, deleted, and updated, respectively. Moreover, the ratios of Javadoc/non-Javadoc ICCs among all comment changes range from 7.0%/7.5% (lower quartile) to 21.7%/15.4% (upper quartile). Only 234 (5.33%)/86 (1.96%) of 4,392 repositories have 0% Javadoc/non-Javadoc ICCs, while only 58 (1.3%) repositories have no ICCs of either type.

We also study the evolution of ICC ratios. Figure 4 shows the ICC ratio by repository age, where each observation is the ICC ratio of all comment changes within a year in the repository. We observe a negative correlation between the ICC ratio and repository age (for Javadoc, Spearman



(a) Javadoc ICCs (b) non-Javadoc ICCs

Fig. 4. ICC ratios by repository age in years.

$\rho = -0.100$, $p < 0.001$; for non-Javadoc $\rho = -0.098$, $p < 0.001$), which suggests a lower ICC ratio as repositories mature.

Finding 2: ICCs exist widely across observed repositories: 98.7% (4,334/4,392) of the repositories have some ICC; ∼15.5% of 24M+ comment changes are ICCs, suggesting a considerable number of suboptimal comments behind the ICCs. Javadoc comments are more likely to be independently added, while non-Javadoc comments are more likely to be independently deleted, suggesting that Javadocs are more likely to be missing or incomplete, while non-Javadoc comments are more likely to be redundant or outdated. The ICC ratio is negatively correlated with the repository age, suggesting a lower ICC ratio as repositories mature.

4 RQ2: WHAT COMMENTS ARE CHANGED INDEPENDENTLY AND HOW? WHAT FACTORS ARE ASSOCIATED WITH THE CHANGES?

4.1 Methodology

Construction of Taxonomy. We conduct a thematic analysis [34] of 3,533 sampled ICCs from Section 3.1. First, two authors conduct inductive open coding separately on randomly selected 30% of the ICCs. We then compare the list of codes and themes to develop a coding guide with definitions and examples for each identified theme. The process is as follows:

- Generate initial codes. For each selected ICC, carefully read the changed comment and its commit (diff and message), to generate the initial codes for three dimensions: (1) **what** is changed (comment); (2) **how** it changed (commenting activity); and (3) when possible, **what factor** is associated with the change (factor). Only a fraction of ICCs have additional information in their commit messages that allow us to identify factors explaining the change, e.g., the commit message in commit `4ab47c63` [24] says *"[j]avaDoc cleanup: useless @see clauses, broken links, non-existent @inhericDoc tags"* which indicates that Javadocs are removed because of outdated or redundant component.
- Group initial codes that have similar key information. Organize all initial codes into themes, suggesting the nature of changed comments, activities of the changes, and factors associated with the changes.
- Define the final themes. Consider each theme (i.e., comment category, commenting activity category, and factor), whether it contains sub-themes, and how these sub-themes interact and relate to the main theme.

Then, three authors (including the former two) use the coding guide to independently analyze the complete set of data. Each ICC is labeled by two authors. ICCs that cannot be classified into any category are put in a new category, named *Pending*. For "what" and "how", we label all 3,533

sampled ICCs. However, most commits have uninformative (e.g., "*Fix javadoc*" [29]), irrelevant (e.g., "*add a test which proves the problem*" [33]), or even empty commit messages, so we can label only 400 ICCs, and label each ICC with *only one* factor. The inter-rater reliability during independent labeling is 0.91 (Cohen's kappa). For conflicted labels, we discuss, and the final judgment is made by a non-author arbitrator who has more than six years of Java experience and conducted similar qualitative analyses before. For ICCs in *Pending*, the arbitrator helps further label the ICCs and determines whether new categories need to be added.

4.2 Results

We derive a three-dimensional taxonomy of ICCs, including categories of comments, categories of commenting activities, and associated factors with changes, as shown in Table 4. The table rows show four categories (with 13 subcategories) of comments that are changed, and the table columns show six categories of commenting activities for the changes. Note that "na" stands for "not applicable". Each cell has two lines. The first line contains the number of ICCs for the combination of the corresponding comment category and commenting activity category. The second line shows the numbers of ICCs that can be labeled by the three factors—Divergence, Tool, and Procrastination—respectively. The third column shows the distribution of comment subcategories in a related study [84], and the forth column shows the distribution in our work (Formatting and Others (FaO) are excluded in the calculation). For each cell, the number before '/' is for Javadoc and after for non-Javadoc. All the numbers add up to 3,763, greater than the number of inspected ICCs (3,533) because one change may involve multiple comment categories, e.g., a newly added Javadoc may contain both Functionality Summary and Usage.

The first two dimensions of our taxonomy—*what is changed* and *how* it changed—are inspired from two earlier studies: taxonomy of source code comments [84] and taxonomy of comment-related changes [112]. We make revisions to their taxonomies based on our context and systematically compare with them in Section 9.1.

The four categories of comments are the following:

- **Code Logic:** Comments that describe the code behavior.
  - **Functionality Summary (FS):** Comments that summarize the code functionality, including the functionality of certain code fragments and the explanation for variables.
  - **Expand:** Comments that provide the context information of how code works, e.g., the condition under which the code enters a particular branch.
  - **Usage:** Comments that describe how to use certain APIs, e.g., information related to method parameters, return value, and exceptions in Javadoc.
  - **Purpose:** Comments that explain why the corresponding code fragment is used or why a certain algorithm is applied.
  - **Code File Structure (CFS):** Comments that describe the structure of the code file and split different parts in a file.
- **Under Development:** Comments that mark work under development or help developers in maintenance.
  - **TODO:** Comments that document unfinished work or unfixed bugs.
  - **Commented Code (CC):** Commented source code, including code that was used for testing/debugging and code examples in Javadoc.
  - **Incomplete:** Comments that provide no useful information, e.g., `@param` tags in Javadoc without explanation of the parameter.
- **Tool Related:** Comments that are related to corresponding tools.
  - **Auto-Generated (AG):** Comments generated by tools or IDE plugins.

Table 4. Three-Dimensional Taxonomy of ICCs

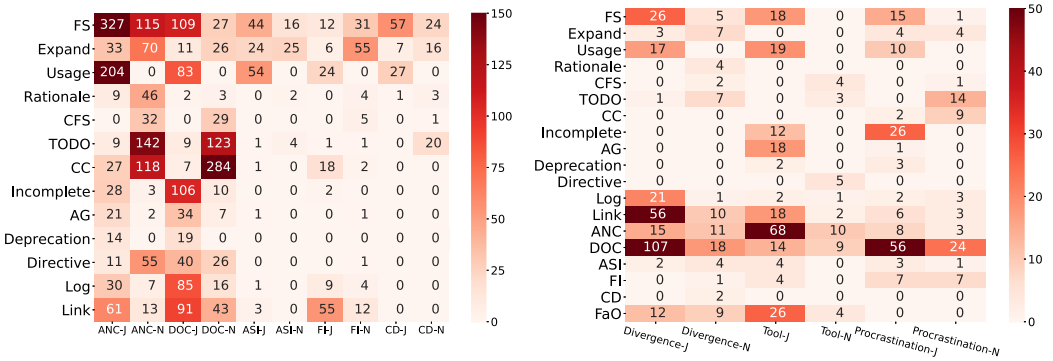
Comment Category	Comment Subcategory	~Freq. in [84]	~Freq. in our work	Commenting Activity Category: ANC	DOC	ASI	FI	CD	FaO
Code Logic	FS	40.5%	24.2%	327/ 115 3+18+2/ 1+0+0	109/ 27 22+0+12/ 3+0+0	44/ 16 1+0+1/0+0+0	12/ 31 0+0+0/0+0+1	57/ 24 0+0+0/1+0+0	Translation 12/ 12 8+0+0/ 8+0+0
	Expand	1.5%	8.7%	33/ 70 0+0+1/ 6+0+1	11/ 26 3+0+2/ 0+0+1	24/ 25 0+0+1/0+0+0	6/ 55 0+0+0/1+0+2	7/ 16 0+0+0/0+0+0	Adjusting Lines and Spaces 279/ 178 4+26+0/ 1+4+0
	Usage	23.9%	12.5%	204/ na 4+14+0/ na	83/ na 12+2+10/ na	54/ na 1+3+0/ na	24/ na 0+1+2/ na	27/ na 0+0+0/ na	
	Rationale	1.9%	2.2%	9/ 46 0+0+0/ 4+0+0	2/ 3 0+0+0/ 0+0+0	0/ 2 0+0+0/0+0+0	0/ 4 0+0+0/0+0+0	1/ 3 0+0+0/0+0+0	
	CFS	0.4%	2.1%	na/ 32 na/ 0+4+0	na/ 29 na/ 2+0+0	na/ na na/0+0+0	5/ na na/0+0+1	1/ na/0+0+0	Updating Mark 0/ 6 0+0+0/ 0+0+0
Under Development	TODO	1.4%	9.9%	9/ 142 0+0+0/ 0+0+2	9/ 123 1+0+0/ 2+3+11	1/ 34 0+0+0/4+0+1	1/ 1 0+0+0/0+0+0	0/ 20 0+0+0/1+0+0	Fixing Typo 34/ 93 0+0+0/ 0+0+0
	CC	2.5%	14.5%	27/ 118 0+0+0/ 0+0+0	7/ 284 0+0+1/ 0+0+9	1/ 0 0+0+1/0+0+0	18/ 2 0+0+0/0+0+0	na/ na/ na	Moving Comment 0/ 5 0+0+0/ 0+0+0
	Incomplete	0.7%	4.8%	28/ 3 0+6+2/ 0+0+0	106/ 10 0+6+26/ 0+0+0	na/ 2 na/ na	0/ na 0+0+0/0+0+0	/ na/ na	
Tool Related	AG	1.6%	2.1%	21/ 2 0+14+0/ 0+0+0	34/ 7 0+4+1/ 0+4+0	1/ 0 0+0+0/0+0+0	0/ 1 0+0+0/0+0+0	na/ na/ na	Noise 0/ 2 0+0+0/ 0+0+0
	Deprecation	0.4%	1.1%	14/ na 0+2+3/ na	19/ na 0+0+0/ na	na/ na na/ na	na/ na/ na	na/ na	
	Directive	7.8%	4.2%	11/ 55 0+0+0/ 0+5+0	40/ 26 0+0+0/ 0+0+0	na/ na na/ na	0/ 1 0+0+0/0+0+0	na/ na na/ na	
Metadata	Log	4.3%	4.8%	30/ 7 6+1+0/ 0+0+0	85/ 16 15+0+2/ 1+1+3	1/ 0 0+1+0/0+0+0	9/ 4 0+0+1/0+0+0	na/ na/ na	
	Link	13.1%	8.9%	61/ 13 2+13+0/ 0+1+0	91/ 43 54+2+2/ 10+1+0	3/ 0 0+0+0/0+0+0	55/ 12 0+3+4/0+0+3	na/ 0+0+0/0+0+0	
Sum		100%	100%	**774/ 603** 15+68+8/11+10+3	**596/ 594** 107+14+56/18+9+24	**129/ 47** 2+4+3/4+0+1	**127/ 116** 0+4+7/1+0+7	**92/ 64** 0+0+0/2+0+0	**325/ 296** 12+26+0/ 9+4+0

For each cell, the number before '/' is for Javadoc and after for non-Javadoc. The three numbers in the second line refer to Divergence, Tool, and Procrastination, respectively.

  - **Deprecation:** Deprecation information in Javadoc, i.e., `@deprecated` tag, the reason why the API is deprecated and the alternative.
  - **Directive:** Comments used by tools, e.g., marker for a static analysis tool to ignore some code.
- **Metadata:** Comments that reveal code metadata.
  - **Log:** Comments that document the author, e.g., `@author`, and version information, e.g., `@since` and `@version`.
  - **Link:** Links in comments, including `@see` and `@link` tags in Javadoc and URL links.

The six categories of commenting activities are the following:

- **Adding New Comment (ANC):** Introducing a new comment.
- **Deleting Obsolete Comment (DOC):** Deleting an old comment.
- **Adding Supplementary Info (ASI):** Adding more information to an existing comment.
- **Fixing Inconsistency (FI):** Fixing an inconsistency between code and comment, including deleting inconsistent comments and revising the outdated part, without adding new information.
- **Clarifying Description (CD):** Updating a comment to restate the expression without introducing or deleting any information.
- **Formatting and Others (FaO):** Changing the format of a comment or performing other minor activities.



(a) The distribution of ICCs among combinations

(b) The distribution of ICCs among factors

Fig. 5. Distributions of ICCs. Each label on the x-axis contains two abbreviations separated with a dash. The first refers to categories of commenting activities and factors in (a) and (b), respectively. The second refers to comment types, where J refers to Javadoc, and N refers to non-Javadoc.

We call a pair of a comment (sub)category and a commenting activity category a *combination*. Figure 5(a) shows the distribution of ICCs among combinations, and some combinations are more frequent than others. Some are more frequent simply because their comments are more frequent among *all comments*, e.g., Functionality Summary and Usage (40.5% and 23.9% in [84]). However, some other combinations are also relatively frequent among labeled ICCs, although their comments are *infrequent* among all comments that reported by Pascarella et al. [84]:

- Some comment categories have a much higher frequency of Deleting Obsolete Comment than Adding New Comment, e.g., Commented Code, Incomplete, and Log. The combinations of Deleting Obsolete Comment with Commented Code and Incomplete are mostly Procrastination; we find that these comments are removed because they are considered old and useless. The combinations of Deleting Obsolete Comment and Log are mostly Divergence; we find that `@author` and `@since` are forbidden in some repositories and removed as further explained in Section 5.2.
- Some categories have a relatively higher frequency of Fixing Inconsistency, e.g., Link and Usage in Javadoc, and Expand in non-Javadoc. These combinations are mostly Procrastinationand changed due to outdated comments, suggesting that these comments may easily become inconsistent with the code but developers do not update them promptly.
- Expand and Usage have a much lower frequency than Functionality Summary in code files [84] and for Adding New Comment in Table 4, while they have a similar frequency as Functionality Summary for Adding Supplementary Info. The comparison may suggest that developers often overlook Expand and Usage when they add new comments, making Expand and Usage more likely to be missing or insufficient.

Finding 3: We create a three-dimensional taxonomy for ICCs, with four comment categories (13 subcategories), six commenting activity categories, and three associated factors with changes. Some combinations of the comment and activity have a relative higher frequency, where the categories of comments tend to be easily overlooked, outdated, or redundant, e.g., Link and Expand.

The third dimension of our taxonomy is completely novel, showing factors associated with comment changes. We derive these factors by inspecting 400 ICCs with informative commit messages in detail (each ICC is labeled with *one* factor), as explained in Section 4.1.

Divergence is derived from all ICCs that change comments to follow a specific convention. Developers with diverse backgrounds may hold diverse opinions on commenting, which leads to ICCs. According to our manual inspection, the diverse opinions on commenting, without a specific convention on commenting, may lead to suboptimal comments that are insufficient, redundant, or inconsistent on formatting and wording. In particular, developers have divergences on several aspects:

- What should or should not be written in comments, e.g., specific API-related information and author tags. Developers may not realize that certain specific information is needed when writing the comment. E.g., Commit `d526bc3b` [26] says *"clarify sync/async for API's"* that adds synchronous information in the comment. Developers may not realize that some components should not be written in comments. E.g., commit `566d851e` [32] says *"removes author tags since they add no value and can be inferred from the git history."* Developers may not realize that some added comments are redundant. E.g., commit `c3514caf` [22] says *"removed some more unnecessary javadoc inherits."*
- Where to comment. Developer may not realize which code is not self-explanatory and requires comment. E.g., Commit `ba80a5ab` adds Javadocs for where the Javadoc is missing in the module with saying *"[e]nsure that the krad-data module is fully javadoc'd."*
- Comment format. Developers may write comments in inconsistent format and later unify the format to follow specific conventions to address the inconsistency. E.g., commit `c7c155a8` [28] says *"reformatted to follow JBoss Community conventions."*
- Use of language. Developers may leave comments in other language and translate them to English later, such as commit `f9715c83` [25] saying *"[t]ranslated comments to English."*
- Commenting maintenance, i.e., using TODO comments or the issue tracker to mark unfinished tasks. E.g., Commit `3094bd0a` [19] says *"[r]emoving "TODO" comments (made them issues in issue tracker)."*

Tool is derived from all ICCs with commit messages that explicitly mention the use of tools. Two kinds of tools are witnessed in ICCs:

- **Comment-checking tools.** Developers use tools to detect or reformat suboptimal comments, e.g., Commit `0283f7ff` [31] says *"[f]ix a pile of Javadoc warnings."* Specific suboptimal comments can be detected by comment-checking tools, such as wrong parameter tags, missing Javadoc, and disorganized format. However, such suboptimal comments may still be introduced due to the absence or ineffective use of comment-checking tools.
- **Comment-generating tools.** Some developers use tools to generate comments automatically, while generated comments are often considered useless and eventually removed, e.g., Commit `06fe9f13` [15] says *"remove auto-generated useless comments"* and removes generated comments in 203 files. Using comment-generating tools at current appears to lead to suboptimal comments.

Procrastination is derived from all ICCs with outdated or duplicate comments. The tendency for developers to delay necessary work can lead to suboptimal comments. We identify several typical scenarios in labeled ICCs:

- Developers fix or remove outdated references and broken links after related code changes, e.g., commit `4ab47c63` [24] says *"[j]avaDoc cleanup: useless @see clauses, broken links, non-existent @inhericDoc tags."*

- Developers remove duplicate comments from packages or from copied code, e.g., commit `1508784d` [23] says *"[r]emove duplicated comments from the packages."*
- Developers fix or remove incorrect comments where corresponding code was already changed, e.g., commit `c52b1bad` [30] fixes dozens of wrong parameter tags.
- Developers prepare or complete Javadoc skeleton, or remove empty Javadocs, e.g., commit `118c7940` [18] says *"[p]repare the project for javadocs."*
- Developers remove TODO comments that are already resolved or abandoned, e.g., commit `eeb59e84` [27] says *"remove stale TODO that has already been taken care of."*
- Developers remove obsolete commented code, e.g., commit `b980b1b2` [16] says *"[c]leaned up old unused code"* and removes 584 lines of commented code.

We label 165 ICCs as Divergence, 134 as Tool, and 101 as Procrastination. Each combination (of comment category and commenting activity category) may have a different distribution of these labels. For example, for Deleting Obsolete Comment and Expand, Javadoc has three ICCs with Divergence, zero ICCs with Tool, and two ICCs with Procrastination (while six ICCs could not be labeled). The distribution of labels in each combination is shown in the second line of each cell in Table 4. Three numbers are the number of Divergence, Tool, and Procrastination labels, respectively.

The factors vary between comment and commenting activity categories of ICCs. The distribution of ICCs among factors is shown in Figure 5(b). For comment categories, Commented Code is mostly Procrastination and Auto-Generated is mostly Tool; and for commenting activity categories, Adding New Comment is mostly Tool and Divergence, Deleting Obsolete Comment is mostly Divergence, and Fixing Inconsistency is mostly Procrastination. The factors also vary between Javadoc and non-Javadoc comments. For example, in the combination of Adding New Comment and Code Logic, Javadoc is mostly Tool while non-Javadoc is mostly Divergence because tools provide support to detect where necessary Javadoc is missing, while non-Javadoc has no similar practical tools (but has some research prototypes [56, 58, 69]).

Finding 4: The three factors associated with changes explain that ICCs are caused by (1) diverse opinions on commenting (Divergence), (2) the use of tools (Tool), and (3) tendency to delay work (Procrastination). The factors vary between Javadoc and non-Javadoc because only Javadoc benefits from tools.

5 RQ3: HOW PREVALENT ARE COMMENTING GUIDELINES? WHAT GUIDELINES ARE SPECIFIED; ARE THEY VIOLATED?

5.1 Methodology

We randomly sample 600 (of 4,392) repositories to ensure a confidence level of 95% and a confidence interval of 5%, following previous work [51, 84, 105, 117], and try to locate any project-specific guidelines for writing comments. For each repository, two authors independently collect all sentences concerning commenting practice in guidelines via manually reading through the `README` files, `CONTRIBUTING` files, wiki pages, and project websites (if provided by the repository on GitHub or mentioned in the `README` file). Then they merge the collected sentences and discuss together to decide if each sentence is indeed a guideline on commenting. All conflicts are resolved by the non-author arbitrator. After merging the results from each author, we find commenting-related sentences for 93 repositories. Following the same procedure described in Section 4.1, we conduct a thematic analysis on all retrieved commenting-related sentences for the 93 repositories, and construct a taxonomy of commenting guidelines. Each repository is labeled with all involved

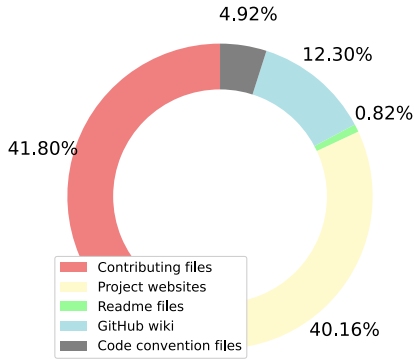


Fig. 6. The sources of identified commenting guidelines.

categories of commenting guidelines. The inter-rater reliability during independent labeling is 0.89 (Cohen's kappa). The conflicts are resolved through discussion, and judged by the non-author arbitrator as in Section 4.1.

To investigate if repositories have violations on commenting guidelines, two authors discuss and select twelve guideline subcategories with clear and consistent definitions that can be checked automatically. We write scripts to check for violations of the guidelines in these subcategories, and run scripts on the repositories with the corresponding guidelines. For each *actively maintained* repository with violations, we open a GitHub issue to obtain developers' feedback about the reasons for violations and the plan for addressing them. We follow the recommendations proposed by Feitelson [40], except for mentioning the purpose. When reporting the issues, we present the violated commenting guidelines with links, explain violations with concrete examples, and ask if developers plan to fix them.

5.2 Results

Prevalence of Commenting Guidelines. We find that 53.7% (322/600) of repositories provide some (coding or commenting) guidelines for developers, but only 15.5% (93/600) provide some guidelines for code comments. We find commenting guidelines of 93 repositories from five sources, including 122 files or web pages. Figure 6 shows the distribution of sources, indicating that commenting guidelines are often documented on contributing files of GitHub or project websites. Even those repositories having commenting guidelines, may still have some violations. One explanation for such a behavior can be developers' diverse opinions on writing comments, which cause suboptimal comments.

Taxonomy of Commenting Guidelines. We explore which elements are included in the commenting guidelines, and provide a taxonomy shown in Figure 7. We discover three categories of elements, such as,*where* to write comments, *what* to write in comments, and *other* commenting guidelines. We find that only 10.0% (60/600) of repositories specify guidelines on *where* to comment, 10.5% (63/600) specify *what* to comment, and 14.5% (87/600) specify either of them. We also find 5.0% (30/600) of repositories have *other* commenting guidelines, i.e., on the readability and maintenance.

We find that guidelines vary across repositories. Different repositories even have conflicting guidelines, e.g., nine guidelines ask developers to leave author tags in comments, while 14 guidelines forbid author tags. As explained in documentations, guidelines forbid author tags for three main reasons: author tags are hard to maintain [64]; author tags promote code ownership, which is considered bad [64]; and author information can be acquired in other ways, e.g., from version-control systems [2].

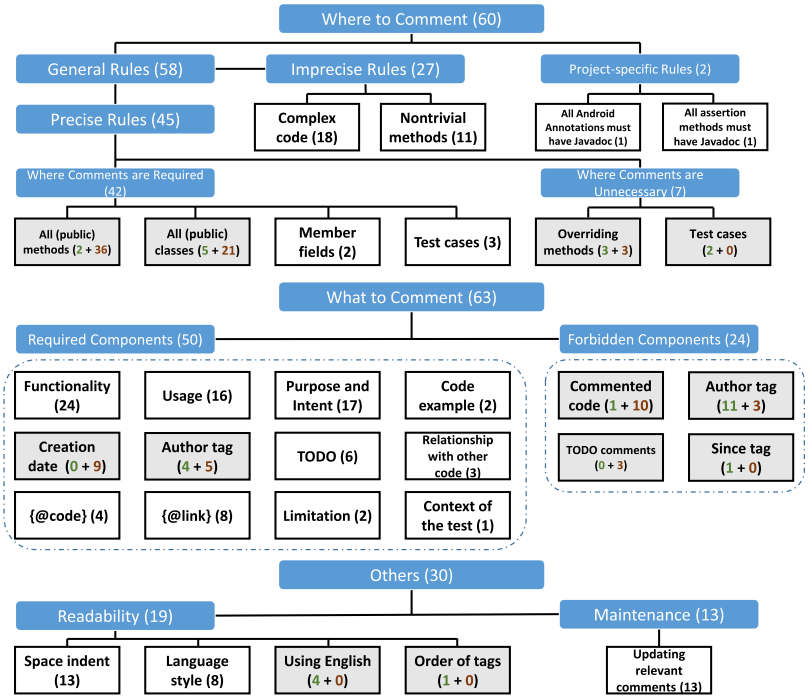


Fig. 7. Our taxonomy of commenting guidelines. The numbers are repositories with specific guidelines. Twelve subcategories with gray background are checked if violations exist, and the numbers in green and red are the numbers of repositories without and with any violations, respectively.

The most common rules that hold across repositories are "where comments are required", and "required components". Guidelines in "where comments are required" specify which code entities should have comments, e.g., all public methods and classes. Guidelines in "required components" specify what should be documented in comments, e.g., the functionality summary and usage information in Javadocs for methods, TODO comments should include related issue link or due date, code token in Javadoc should be wrapped by `@code`. Correspondingly, there are also guidelines to forbid where to comment and what to comment. Overriding methods and test cases are considered unnecessary to comment in seven repositories, and commented code, TODO comments, author tag, and since tags are forbidden in 24 repositories. There are also two project-specific rules that are associated with the purpose of repositories, i.e., all Android Annotation in `androidannotations/androidannotations` and assertion methods in `joel-costigliola/assertj-core` should have comments. Detailed explanations for each category in Figure 7 can be found in the code book in our supplementary material [100].

Some repositories have imprecise rules, e.g., 27 repositories state that "complex code" and "nontrivial methods" should have comments. However, these guidelines can be hard to follow as different developers may interpret "complex" and "nontrivial" differently.

Finding 5: Commenting guidelines are missing in most repositories. We establish a hierarchical taxonomy of commenting guidelines with 31 subcategories. Only 14.5% of repositories have commenting guidelines on "where" or "what" to comment.

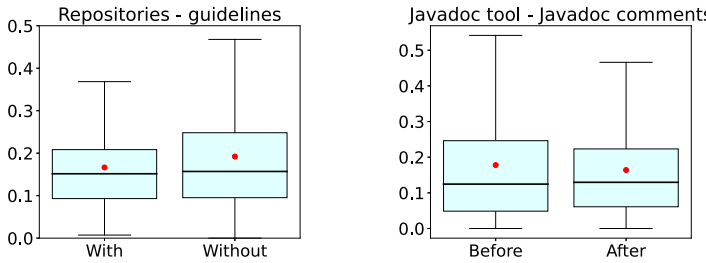


Fig. 8. The ICC ratio among repositories (a) with or without commenting guidelines; and (b) before and after the use of tool Javadoc.

Impact of Commenting Guidelines. Figure 8 shows the distribution of the ICC ratio of repositories with and without commenting guidelines. To check if "where" or "what" guidelines can alleviate the problem of ICCs, we run the Welch one-tailed two-sample t-test. The results show that the ICC ratios of the repositories with "where" or "what" guidelines are statistically significantly (at $\alpha = 0.05$) lower than those of the repositories without commenting guidelines ($t = -2.009$, $p = 0.034$). The results indicate that the lack of "where" and "what" guidelines may aggravate the problem of ICCs. We also find that when including "others" guidelines in the same test, the result is no longer significant ($p = 0.370$), suggesting "others" guidelines are not effective enough.

Finding 6: The presence of commenting guidelines on "where" or "what" to comment is statistically associated with a lower ICC ratio, suggesting that the lack of such commenting guidelines has a negative impact on the commenting practice, i.e., developers tend to submit more ICCs without guidelines.

Violations of Commenting Guidelines. Figure 7 shows with grey background the twelve subcategories for which we check violations, and the numbers in green and red are the numbers of repositories without and with any violations, respectively. We find that 67.7% (8/12) of above subcategories have violations, and 85.5% (59/69) of repositories with such commenting guideline subcategories have violations. To understand the developers' attitude to these violations, we open issues for all 24 (of 59) actively maintained repositories with an issue tracker. In these issues, we list the repository's commenting guidelines and the violations. We have received responses from developers for 13 issues. According to the responses, most (11/13) intend to change (some of) the code to correct the violations, one response indicates that the commenting guidelines should be revised, and one response indicates that both code and guidelines should be changed. To justify why some violations are present, or why some violations can be ignored, developers provide two main reasons: one is that commenting guidelines are relatively new to some repositories, while parts of the code are old and predate the commenting guidelines; and the other is that some commenting guidelines are idealized, e.g., requiring 100 percent Javadoc coverage on public methods, while some simple methods are self-explanatory and not worth commenting.

Finding 7: 85.5% of the studied repositories that have commenting guidelines have some violations. Comments predating guidelines or idealized guidelines tend to cause the violations.

6 RQ4: HOW PREVALENT IS THE USE OF TOOLS TO ASSIST COMMENTING PRACTICE? ARE THE TOOLS EFFECTIVE?

6.1 Methodology

We explore two kinds of tools: comment-checking tools and comment-generating tools. We investigate the tools' configurations and related ICCs, and conduct a survey of active developers to answer the question.

6.1.1 Investigation of Tool Configuration and Related ICCs. For comment-checking tools, we study the use of `Javadoc` and `Checkstyle` because (1) these two tools are the most popular to generate API documentation and check coding style (including commenting style), respectively; (2) they are often mentioned in commit messages; and (3) the use of these tools can be automatically detected (unlike the use of IDE plugins or comment-generating tools).

To investigate the prevalence of `Javadoc` and `Checkstyle`, we check if each of 4,392 repositories has a build file (`pom.xml` or `build.gradle`) that includes these tools, or a configuration file for `Checkstyle` (`*checkstyle.xml`). We further collect all checks from all `Checkstyle` configuration files so we can analyze to what extent they have comment checks. If a repository has more than one configuration file, we would take the concatenation of all configuration files.

To understand if comment-checking tools are effective in assisting commenting practice, we seek patterns on how developers use these tools, and if the introduction of certain tools have an impact on commenting practice. On the one hand, we manually inspect all commits containing sampled ICCs that use comment-checking tools, and try to figure out if these commits follow certain patterns. We also check what kind of suboptimal comments can be detected by comment-checking tools, and how developers address them. On the other hand, we compute the ICC ratios among comment changes before and after introducing the tool to investigate if the introduction of tools can help commenting practice. For each repository that we find using considered tools, we compute when it first started using the tool by finding the earliest commit related to the relevant file. We excluded repositories with fewer than ten comment changes before or after the introduction. We then conduct a paired one-tailed t-test on the ICC ratios before and after the introduction of tools.

For comment-generating tools, the use of these tools is hard to identify automatically. Thus, instead of investigating the prevalence and effectiveness of comment-generating tools directly, we target generated comments. We manually inspect all 66 ICCs that belong to Auto-Generated (discovered by explicit commit messages or apparent patterns) in our sample, and identify two tools that developers used to generate comments. Two authors inspect all generated comments in above ICCs together and iteratively merge comments with similar content. Eventually, they identify four categories of generated comments, with each assigned description of the content. They further check the documentation [3, 61] of two identified tools and find that examples of generated comments within the documentation fit the identified categories. We then investigate the amount and the life cycle of Javadoc skeletons because they are likely generated by tools and can be used as a proxy to estimate the prevalence and effectiveness of generated comments.

6.1.2 Survey to Developers. We conduct a survey to further elicit developers' opinions on comment-checking and comment-generating tools. We follow the principles of Dillman et al. [37] to design the survey. Based on the investigation of two kinds of tools, we design a questionnaire that asks how often do developers use tools, how do they use tools, and if they are satisfied with current tools. We first conduct a pilot study with four researchers with experiences on code comments and open-source practices, and three software engineers with more than five years of programming experience. According to their feedback, we add skip logic (i.e., respondents that did not use or witness the use of tools will skip corresponding following questions), and eliminate questions that respondents had difficulty answering. The final questionnaire includes

Table 5. Questions in the Survey

	Questions
Q1	Have you yourself used comment-checking tools or witnessed someone else use them, e.g., Javadoc and Checkstyle? (Single-answer question)
Q2	How often do you (or other developers in your projects) ignore warnings (or even errors) reported by comment-checking tools? (Single-answer question)
Q3	Why do you think developers ignore the warnings (or even errors)? (Multiple-answer question)
Q4	Have you yourself used comment-generating tools or witnessed someone else use them? (Single-answer question)
Q5	How often do you (or other developers in your projects) update auto-generated comments? (Single-answer question)
Q6	Do you think current comment-generating tools generate good comments? (Single-answer question)

three background questions for demographic purpose, six choice questions, and an open-ended question to collect respondents' additional insights and experience beyond the choices we provide. Table 5 shows the six choice questions that we asked in the survey.

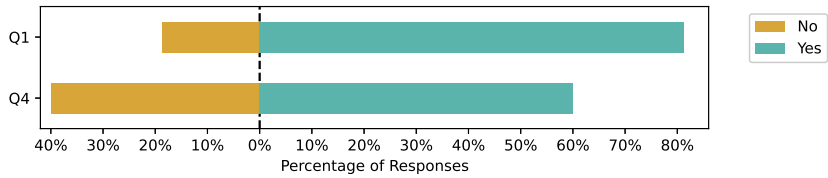
To find potential participants with recent contributions to open-source communities who are familiar with current commenting practices, we select developers who have contributed more than ten commits in the last year. From the repositories we collect, we obtain 6,059 developers and their email addresses. We randomly sample 1,000 addresses and send them emails containing our online questionnaire's link. Out of 1,000 emails, 132 could not be delivered. The survey ran for two weeks, and we received 80 valid responses eventually. The response rate is 9.2%, comparable to the response rate (6%–36%) in other surveys in software engineering studies [95, 107, 116].

6.2 Results

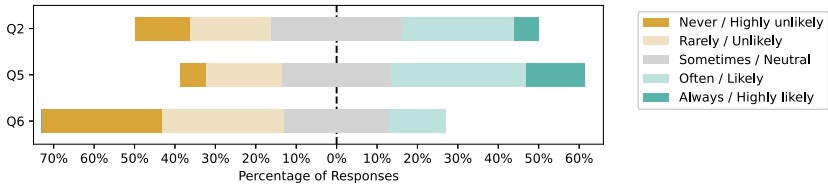
6.2.1 Comment-Checking Tools. We find that comment-checking tools are widely used by developers: 59% of (2,599/4,392) repositories incorporate `Javadoc` in a build file. We also find 28% of (1,253/4,392) repositories use `Checkstyle`, where 1,025 repositories incorporate `Checkstyle` in a build file, and 694 repositories have a configuration file. According to the survey, 81.2% of respondents have used or witnessed the use of comment-checking tools, again indicating the wide use of comment-checking tools.

In the 694 `Checkstyle` configuration files, our analysis of the Checkstyle configuration rules identifies 24 comment checks, e.g., `JavadocMethod`, `JavadocStyle`, `AtclauseOrder`, and `TodoComment`. 60.5% (420/694) of these files have at least one comment check. We find that all top ten frequently used checks examine Javadoc, including `JavadocMethod`, `JavadocType`, and `JavadocStyle`. Only one check (`TodoComment`) can also examine non-Javadoc. Moreover, commenting related checks in `Checkstyle` mainly examine the style of comments, or if specific Javadocs or tags exist.

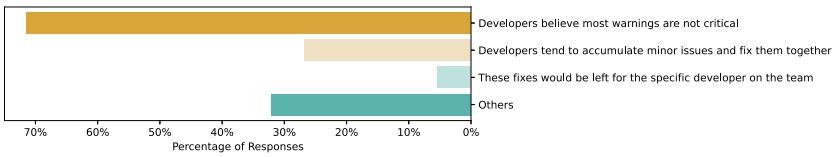
Our investigation on the ICCs labeled with Tool discovers the pattern that suggests that developers often ignore output from comment-checking tools, leaving comments suboptimal. We discover that developers tend to fix inconsistencies in Usage and Link in batch, e.g., addressing warnings and errors from comment-checking tools like `Javadoc` and `Checkstyle`, instead of fixing the inconsistencies timely in the same commit when the code is changed. Of all ICCs labeled with *Tool*, 73.1% (98/134) are related to comment-checking tools, which add Javadocs for methods, add missing `@param` and `@return`, fix broken `@see` and `@link`, and reformat comments. Of these 98 ICCs, 86.7% (85/98) belong to commits with more than ten files changed, and 58.2% (57/98) belong to commits with more than 100 files changed. (Only 57.0% of sampled 3,533 ICCs belong to commits with



(a) Responses to Q1 and Q4



(b) Responses to Q2, Q5 and Q6



(c) Responses to Q3

Fig. 9. Responses to the survey.

more than ten files changed, and 26.2% for 100 files.) The results of the survey provide evidences that developers tend to overlook warnings from comment-checking tools. As shown in Figure 9, 33.8% of respondents often or always ignore reported warnings, 32.3% report "sometimes", 20.0% report "rarely", and 13.8% report "never". As for the reason why some warnings are ignored, 71.4% of respondents think that developers believe most warnings are not critical, and 26.8% of respondents think that developers tend to accumulate minor issues and fix them together, which fits the pattern we observe in related commits.

The statistical analysis shows a statistically significant effect on the ICC ratio for the introduction of `Javadoc` but not for `Checkstyle`. We obtain the paired ICC ratios of 1,691 repositories using `Javadoc` and 897 repositories using `Checkstyle`. For the `Javadoc` tool, the results (for Javadoc comments, $p = 0.004$; for non-Javadoc, $p = 0.186$) show statistically significant differences of Javadoc ICC ratio before and after the use of `Javadoc`, as shown in Figure 8, suggesting that `Javadoc` is likely to help commenting practice. For `Checkstyle`, the results (for Javadoc, $p = 0.306$; for non-Javadoc, $p = 0.471$) are *not* statistically significant, which may suggest the ineffective use of `Checkstyle` to check comments.

Finding 8: Comment-checking tools are widely used: 59% (2,599/4,392) of projects incorporate `Javadoc` in their build files and 28% (1,253/4,392) incorporate `Checkstyle`. However, developers often overlook warnings raised by these tools and tend to fix the warnings in batch, therefore leaving suboptimal comments neglected for long periods of time. A statistical analysis shows a significant effect on the ICC ratio for the introduction of `Javadoc` but not for `Checkstyle`, indicating that `Checkstyle` by itself may not help the comment quality.

6.2.2 Comment-Generating Tools. We identify four kinds of comments that can be generated by comment-generating tools. While constructing our ICC taxonomy, we notice mention of two tools to generate Javadocs: `Apache Maven Javadoc Plugin` [3] and `JAutodoc` [61]. Combining manual inspection of generated comments in ICCs and checking on the documentation of two tools, generated comments include these kinds: (1) **Type information** (generated by `Apache Maven Javadoc Plugin`) only lists method parameters, return values, and their types; (2) **Literal meaning** (generated by `JAutodoc`) provides the information as in Templates and also adds a method description that simply restates the method name; (3) **Log information** such as author and version information; and (4) **Empty Javadoc or Javadoc skeleton** has no descriptive information.

We further analyze the number and the life cycle of empty and skeleton Javadocs, because this kind of comment can be easily identified. Our analysis on empty and skeleton Javadocs discovers that developers add them, but rarely complete and often delete them: (1) For the 28 Adding New Comment/Incomplete Javadoc comments in Table 4, we find only five (17.9%) have been supplemented with more information, while 23 (82.1%) still remain uninformative or were deleted uncompleted. (2) Many Javadoc skeletons are deleted uncompleted. We analyze all 201,518 deleted Javadoc ICCs and identify 11,263 empty Javadocs and 27,172 skeletons with empty Usage tags, which account for 19.1% of all deleted Javadoc ICCs. Javadoc skeletons existed for a long time before they were finally deleted. Half of the above ~38k removed Javadoc skeletons existed for more than one (431 days). (3) Many skeletons still remain. We find 195,525 empty Javadocs and 195,759 skeleton Javadocs with empty Usage tags in the versions of 4,392 Java repositories with a script. These ~400k empty or skeleton Javadocs account for ~2.5% of all ~16M Javadocs.

Our manual inspection on related ICCs also finds that developers are not satisfied with some generated comments. Comments generated by these comment-generating tools are often deleted as useless. In Table 4, 37 Auto-Generated Javadocs and seven non-Javadoc comments are deleted as useless, based on the commit messages for these changes. For example, the commit message for `06fe9f13` [15] in `AKSW/RDFUnit` says "*remove auto-generated useless comments*", and the commit deletes comments in 203 files.

According to the survey, comment-generating tools are widely used by developers. As shown in Figure 9, 60% of respondents report that they have used or witnessed the use of comment-generating tools. However, generated comments often require manual improvement. 47% of respondents often or always update generated comments and only 22.9% respondents rarely or never update them. Only 14.6% of respondents think comment-generating tools can generate good comments, while 58.3% think they cannot, and 27.1% choose to stay neutral. The results indicate that developers are not satisfied with comments generated by comment-generating tools.

Finding 9: Most comment-generating tools only generate uninformative templates, which are often left neglected and eventually deleted. The survey shows that only 22.9% respondents rarely or never manually update generated comments and 58.3% of respondents hold a negative attitude toward current comment-generating tools.

7 IMPLICATIONS

7.1 Research

(1) Comment Prediction: We find that guidelines on where or what to comment significantly reduce the ICC ratio (Finding 5). However, some repositories only have imprecise guidelines, e.g., "complex code" should have comments. These guidelines are hard to enforce due to the divergent opinions of different developers. We could alleviate this problem if we can predict where comments could help. Recent work [56, 58, 69] attempted to address this problem using deep learning models

to predict the position of comments in code files. We recommend more work in this direction and suggest predicting not only where to comment but also what to comment, e.g., the comment categories, so developers can better understand why and what to comment.

(2) Comment Generation: Current comment-generating tools only target Javadoc comments (Finding 9), and research work [7, 13, 54, 55, 65, 72, 96, 111] on comment generation mostly targeted the method granularity as well, especially Functionality Summary. In contrast, Huang et al. [57] presented a learning model to generate block comments. Chen et al. [12] found that state-of-the-art code summarization models perform well for Functionality Summary but not for Expand and Purpose. We find that Auto-Generated comments in sampled ICCs only contain the Functionality Summary that restates the method signature, including parameters and return type (Finding 9). However, our taxonomy of ICCs indicates that other comments, e.g., Usage, are also often added and supplemented (Finding 3). We encourage researchers to further explore comment generation at a finer granularity (non-Javadoc comments), and to target more types of comments (e.g., Usage and Purpose).

(3) Comment Quality Assessment: Based on the identified commenting guidelines in Section 5.2 and the documentation quality attributes [118], we can roughly summarize developers' expectation on comments as below: *Coverage* (comments should cover all necessary code entities); *Significance* (complex and nontrivial code should have comments, while self-explained code not); *Completeness* (comments should provide all necessary information); *Usefulness* (comments that are not useful and easily outdated should be avoided, e.g., commented code); *Consistency* (comments should follow a consistent format and information organization that developers can easily retrieve the information); and *Up-to-date-ness* (comments should be updated promptly with relevant code changes). These dimensions of comment quality illustrate what developers really concern in practice. However, current studies on comment quality mostly focused on *significance* [1, 97] and *up-to-date-ness* [67, 81]. Thus, we recommend researchers to further explore how to comprehensively assess comment quality. For example, for *Completeness*, it might be interesting to investigate if existing comments already cover all valuable information or which categories of information should be supplemented, that requires deep understanding on both comment and code semantics.

7.2 Practices of Commenting

(1) Formulating Commenting Guidelines: Our study shows that most open-source communities lack commenting guidelines, which has a negative impact on maintenance (Finding 7). Only 15.5% of 600 repositories we check have any commenting guideline. We find that guidelines on *where* or *what* to comment decrease the ICC ratio statistically significantly. Therefore, we recommend communities to formulate their own commenting guidelines with both *systematic principles* and *actionable rules* based on their evaluation on comment quality as summarized above. For systematic principles, we advise project maintainers to suggest where to comment (*Coverage*), what should or should not be commented (*Completeness* and *Usefulness*), avoiding uninformative comments (*Significance*), and checking if comments are updated timely in the process of code review (*Up-to-date-ness*). Systematic principles are potentially established with repository-specific differences, e.g., some projects may require detailed Functionality Summary and Usage information for every public method, and some projects only for API in particular modules. According to the feedback on our issues (Finding 7), these principles should be pragmatic, e.g., projects may not ask for all methods to have Javadoc if some simple self-explanatory methods exist. For actionable rules, we advise the following: add the due date in TODO (*Completeness*), do not commit incomplete skeletons or `@author` tags (*Usefulness*), carefully consider commented code (*Usefulness*), and provide the template for Javadoc (*Consistency*).

(2) Enforcing Rules with Tools: Formulating commenting guidelines is important, as well as using tools to enforce rules. We encourage projects to use automatic tools during the code-review process to reduce suboptimal comments. For example, tools that can identify Javadoc skeleton can be used in the code-review process to prevent incomplete comments from being introduced. We identify a set of comment smells that cause maintenance problems and can be detected by the existing tools (e.g., `Javadoc` or `Checkstyle`). In particular, broken links in `@see` and `@link` tags, and inconsistent `@param` and `@return`, can be detected by these tools. However, developers often ignore warnings from these tools (Finding 8). Given that these tools are highly configurable, we advise developers to configure the tools to specifically check these smells and integrate them into the CI/CD pipeline. Moreover, according to the feedback on our issues (Finding 7), when formulating new commenting guidelines, we advise developers to check not only new commits but also old comments to avoid violations.

7.3 Tools

(1) Improving Comment-Checking Tools: Our taxonomy of commenting guidelines indicates various concerns on comments from developers, while not all rules can be automatically checked by current comment-checking tools. We therefore encourage tools to be improved for comment-checking. In particular, tools can add checks to detect redundant comments, e.g., comments that restate the method name and parameter name such as some generated comments. These comments have a high coherence coefficient with code [97] but provide no additional information. We find that multiple categories of comments are required by developers, e.g., Functionality Summary, Usage, and Purpose are frequently asked for Javadocs, while current tools do not check whether such semantic information exists. Thus, tools may be improved to classify comments in a sentence granularity and check if required comments exist. We find that developers tend to avoid comments like Commented Code and TODO because they are useless and hard to maintain. Moreover, we identify considerable empty or skeleton Javadocs in the code files which appear to be useless. Therefore tools may add checks for useless comments like Commented Code. Our taxonomy shows that Javadocs benefit from tools, while non-Javadoc do not (Finding 4). We also find that only Javadocs have a statistically lower ICC ratio with the use of comment-checking tools (Finding 8). Thus, tools may add more checks for the presence, quality and semantics of non-Javadoc. For presence, tools may provide checks for specific complex code, e.g., multiple nested loops. For quality and semantics, tools may provide checks to detect outdated comments inconsistent with the code logic, e.g., Nie et al. [75] proposed a tool to detect outdated TODO comments.

(2) Improving Version-Control System (VCS): Our study finds that many commenting guidelines forbid Log information because it is hard to maintain and replaceable by VCS (Finding 7). However, the most widely used VCS, `git`, cannot fully replace some tags, e.g., `@since` (required by 9 commenting guidelines), despite their potential usefulness. `git` lacks structural understanding of code, e.g., `git blame` only tracks the last modification by line and cannot easily tell when a code entity (e.g., method) is introduced and by whom—`git` cannot track such code ownership and version information precisely. Thus, we encourage tool designers to improve VCS (or develop command level interfaces) to help recover the Log information that is sometimes in comments. Prior studies proposed tools [6, 14, 92] to visualize the evolution of software in different ways, e.g., coloring code lines and drawing node-link diagrams, and tools [36, 74] to track code elements and not just files. Recent work on CodeShovel also shows additional promising results [49] that provided method-level source code histories via mining Git histories with different metrics.

8 THREATS TO VALIDITY

Internal Validity: The ICCs that our technique extracts from hunks may still be related to some code changes far away from the hunk, even in another code file. Also, we may miss comment changes that are actually ICCs but excluded due to nearby irrelevant code changes. Accurate matching of comment and code changes is an open problem [11, 112], and no reliable way exists to detect whether a comment change is related to a distant code change. Therefore, our technique only considers nearby code, and we adopt matching heuristics that align with current best practices, e.g. Javadoc of a method concerns the method body. Our heuristics is shown effective with both high precision (96.4%) and recall (90.0%). The manual inspection of 3,600 automatically retrieved ICCs finds only 67 false ICCs, again indicating our technique is highly precise.

The ICC ratio may not accurately reflect the quality of commenting practice. There are many aspects of commenting practice and many different ways of assessing commenting practice. In this study we investigate how suboptimal comments are produced using ICC as a proxy, so we refer to whether developers add/delete/update comments timely with corresponding changes. Meanwhile, different projects have diverse practices, e.g., some developers may commit the code and corresponding comments separately in a pull request, which has impact on the ICC ratio. To mitigate the bias, we collect ICCs in a commit granularity based on the best practice recommended by Git that "[a] commit should be a wrapper for related changes" [9] and comment changes intuitively should be committed with corresponding code changes. Moreover, our results show that indeed most ICCs improve comments after the change (92.9% of 3,533 ICCs). We find no other suitable metrics that can be applied, e.g., comment density [4, 52], length of comments [97], Flesch reading ease score [39], and textual similarity between source code and source code [71, 97], which only measure a certain dimension of comments like prevalence and readability. Thus, we consider it sensible to use the ICC ratio to evaluate the commenting practice considered in this study, i.e., if developers commit comment changes timely with corresponding code changes.

The definitions of our ICC taxonomy and commenting guideline taxonomy may contain ambiguities, and the manual labeling is also prone to errors and conflicts. To mitigate such threats, we include three authors and one arbitrator in this study. Furthermore, we iteratively discussed and refined taxonomy definitions and labeling criteria through the analysis with sampled 30% data to ensure agreement and reproducibility in the final taxonomy and labeling process.

We cannot guarantee completeness for the three identified factors associated with the changes, because most ICCs (3,133/3,533 in our manual inspection) have no sufficient information in commit messages for inferring reasons, and some straightforward combinations of ICCs require no reasons. For example, for Clarifying Description, we can obtain reasons for only two out of 136 ICCs. Also, 127 ICCs fix typos and require no deep insights for reasons behind typos, thus we obtain no reason for those ICCs. We leave further investigations of this problem for future work.

External Validity: Our study is based on a set of popular open-source Java projects from GitHub, which may not be representative of all open-source Java projects. To mitigate this threat, we adopt selection criteria from prior studies [51, 53, 112] to select projects with diverse sizes and domains. Still, the results of our study may not generalize to proprietary Java projects as they often adopt different practices compared with open-source projects. Communities for other programming languages—such as Python, C, or JavaScript—may have different commenting practices. However, some of our implications still appear to be generalizable, e.g., the commenting guidelines.

Our study only investigates commenting guidelines specified by projects and ignores standard guidelines (such as the Google style guide [60] and Oracle style guide [77]). However, our investigation of standard guidelines shows that commenting guidelines in standard guidelines often concern syntax and formatting, and all of them can fit in our taxonomy of commenting guidelines.

Moreover, standard guidelines often provide general guidelines. Thus, even repositories that follow standard guidelines often specify detailed commenting guidelines (about 40% of repositories that follow standard guidelines in our dataset specify additional commenting guidelines). To conclude, we can still guarantee the completeness of our results.

Our study only investigates `Javadoc` and `Checkstyle`, that may not represent all the comment-checking tools. To mitigate this threat, we select the tools that are frequently used by the sampled repositories—`Javadoc` and `Checkstyle` are used by 59% and 28% of the sampled repositories, respectively. Moreover, we complement our results through conducting a survey with experienced developers, who likely use tools beyond `Javadoc` and `Checkstyle`. To conclude, our results can reflect the overall situation of tools used for commenting, though we only investigate two tools.

9 RELATED WORK

The work most related to ours fits in four groups: comment classification, comment code co-evolution, contributing guidelines, and automatic static analysis tools.

9.1 Comment Classification

Several studies have explored the nature of some comments and established a taxonomy for the explored comments. Padioleau et al. [78] proposed a taxonomy for *comments in operating system code* to guide development of comment-checking tools. Haouari et al. [50] investigated *developers' commenting habits* in Java and defined a taxonomy of comments based on comment object, type, style, and quality. Maalej and Robillard [70] defined a taxonomy of knowledge patterns in *API reference documentation* by inspecting Javadoc comments sampled from JDK. Steidl et al. [97] proposed a taxonomy of comments for further qualification, while they classified comments based on the location (e.g., *"Copyright comments"* and *"Inline comments"*) and functionality (e.g., *"Task comments"*), and we classified comments based on their content. Zhang et al. [115] proposed a taxonomy of *Python comments* for automatic classification, and our taxonomy expands it with more categories (e.g., Commented Code and Deprecation) that cannot fit in their taxonomy. Shinyama et al. [93] proposed a taxonomy of *Java comments* based on the relationship between the comment and corresponding code (e.g., *"Postcondition"* and *"Value Description"*), which is different from our taxonomy that is based on the content. Zhai et al. [114] proposed a two-dimensional taxonomy of comments based on *code entity* and *content*, which better facilitates automated classification, propagation, and program analysis, but is too coarse-grained for obtaining empirical understanding. Geist et al. [44] applied different approaches—heuristics, machine learning, and deep learning—for comment classification and found that machine learning outperforms the heuristics. Rani et al. [89] proposed taxonomies of *class comments in Java, Smalltalk, and Python*, and made comparisons between different languages. Their taxonomy of Java comments follows the one proposed by Pascarella et al. [84], which we make a detailed comparison as follows.

The comment classification studies that are most relevant to us are taxonomies proposed by Pascarella et al. [83–85] and Wen et al. [112]. Pascarella et al. [83–85] proposed a taxonomy of *Java comments* by manually classifying comments sampled from six Java open-source projects, five open-source Android apps, and eight industrial projects. Wen et al. [112] proposed a taxonomy of code-comment inconsistencies fixed by developers. The difference between our taxonomy and their taxonomies is discussed as follows:

- We define comments and commenting activities as two independent dimensions, while Pascarella et al. [84] propose a one-dimensional taxonomy focused only on comments (no activities, derived by inspecting code files), and Wen et al. [112] propose a hierarchical taxonomy that mixes both comments and activities (derived by inspecting commits that focus on comment changes, as identified by keywords in commit messages).

- We derive our taxonomy from fine-grain (hunk-level) classification of ICCs; we inspect more cases (3,533 ICCs vs. 2,000 source code files by Pascarella et al. [84] vs. 362 commits by Wen et al. [112]), with a dataset *more* representative of all commits.[4]
- Compared with the taxonomy proposed by Pascarella et al. [84], we make revisions to comment categories, which include:
  - Expanding existing categories, e.g., expanding "Ownership" with "Log" to include version information like `@since` and `@version`, expanding "Pointer" with "Link" to include URL links, and expanding "Formatter" with "Code File Structure" to include meaningful code file structure splits.
  - Reorganizing categories, e.g., merging their "Exception" with "Usage".
- Compared with Wen et al. [112]'s hierarchical taxonomy of comment and activity combinations (that labeled each of 362 commits with only one combination, our taxonomy uses two independent dimensions. We cannot directly map all taxonomy elements, but some of their root categories are similar to some of our cases, e.g., their "Application Logic" (37.57%, 136/362) to our Code Logic (44.27%, total 1,564/3,533), their "Code Design/Quality" (22.10%, 80/362) to our Under Development (26.78%, total 946/3,533), and their "Formatting and Others" (32.87%, (63+56)/362) to our Formatting and Others (17.58%, (325+296)/3,533).

Overall, compared to all prior work, we study *ICCs* and propose a *three-dimensional taxonomy* in which the *comment* and *commenting activity* dimensions built on prior work [84, 112], and the associated factors with changes is a completely novel dimension. Our taxonomy of the comment dimension differs from previous work in following aspects: (1) our comment taxonomy target all types of Java comment (both Javadoc and non-Javadoc) [70, 78, 89, 115]; (2) our comment taxonomy classifies comments based on the content [50, 93, 97]; and (3) our taxonomy expand existing taxonomies with more informative categories [83–85, 89, 114, 115].

9.2 Code-Comment Co-Evolution

The research on code-comment co-evolution focuses on three aspects: to what extent comments evolve with code [41, 42, 59, 63], how to detect outdated or inconsistent comments [66, 75, 80, 91, 98, 99, 102–104, 119], and how to update comments automatically [67, 82].

Fluri et al. [41, 42] found (12+ years ago) from seven open-source Java projects that 3–10% of the comment changes did not occur in the same *released version* as the associated code changes. Meanwhile, we find that the average ICC ratio, with no associated code changes in the same *commit*, is ~15.5% in 4,392 open-source Java repositories, indicating the prevalence of suboptimal comments. Our results better represent the current situation in open-source development.

Tan et al. [102] proposed iComment to extract synchronization-related implicit rules in comments and check if they match the logic of code within the same C function. Follow-up work on Java includes @tComment [104] that checked whether description of parameters and exception matches corresponding description in Javadoc comments, Toradocu [48] that focused on exceptional behaviors, JDoctor [10] that checked more semantic content from the Javadoc, and upDoc [99] that built code-comment correspondence to ensure that code changes match comment changes. Liu et al. [67] and Panthaplackel et al. [81] proposed approaches to automatically update comments or detect inconsistencies based on the corresponding code change and old comment, using deep learning models learned from a large number of code-comment co-changes. Nie et al. [75]

[4]In Wen et al.'s dataset [112], 54.6% of the commits modify only one file, and 78.0% modify fewer than five files. In comparison, the distribution of commits in our dataset (10.7% modify only one file, 26.4% modify fewer than five files) is more similar to all commits that modify some comments (19.4% modify only one file, 48.0% modify fewer than five files).

proposed a framework to detect which todo comments, written in a specific format, become obsolete when code is completed.

Our study complements empirical knowledge on suboptimal comments, helping to better understand the reasons for code-comment inconsistencies.

9.3 Contributing Guidelines

A few studies explored the contributing guidelines for open-source projects that often include commenting guidelines. Elazhary et al. [38] analyzed `CONTRIBUTING.md` and `README.md` files in 72 projects, finding five categories of contributing guidelines. Related to comments is only their `Contribution Documentation` subcategory of `Pull Request Acceptance Criteria`, reported to exist in 47.2% of 72 projects but provide no further information. Bafatakis et al. [5] and Simmons et al. [94] checked the compliance of Python code to coding standards, including standards on commenting, and found the differences on coding standards between projects and common violations of coding standards in StackOverflow code snippets. Our study complements their work by revealing the distribution of commenting guidelines in different repositories and confirming the violation in code repositories instead of StackOverflow code snippets. Rani et al. [88] identified topics that developers discuss about comments on StackOverflow and Quora, and listed comment conventions extracted from answers. In their following work [87, 90], they investigated conventions of class comments in Java, Python, and Smalltalk, classified commenting convention in a coarse granularity (e.g., *"Formatting" and "Content"*), and manually checked violations in sampled comments. Our study complements their work with the investigation on more repositories (600 open-source repositories instead of six projects) and a comprehensive taxonomy of commenting guidelines that explicitly indicates what rules developer specify for both class comments and inline comments.

We are the first to provide a comprehensive taxonomy of elements in commenting guidelines. We collect commenting guidelines from all online documents in 600 sampled repositories. We contribute an extensive study of commenting guidelines, including the prevalence of commenting guidelines, a taxonomy of elements in commenting guidelines, the extent of impact of commenting guidelines on practice, and the enforcement of commenting guidelines in projects (such as violations and maintainers' responses to reported violations).

9.4 Automatic Static Analysis Tools

Several studies have explored how **automatic static analysis tools (ASATs)** are used by developers in the code review, which include tools like `Checkstyle` that can be used to check comments. Beller et al. [8] investigate the prevalence and configuration of ASATs. Their results show that ASATs were widespread in 2016, but most repositories do not enforce their use strictly, only use the default configuration, and hardly change the configuration. Panichella et al. [79] investigate how ASATs are used in the code review, and found that while false-positive warnings might be raised, the removal of certain warnings before the submission can reduce the amount of effort in code review. Vassallo et al. [110] further explore this scenario and find that different warning categories receive different levels of attention depending on the development context. Zampetti et al. [113] investigate how ASATs are adopted in continuous integration, and find that build failures are often caused by checks related to code standards and missing licenses, instead of potential bugs or vulnerabilities. Our study complements knowledge on how ASATs are used in the commenting practice and how the use of ASATs affects the ICC ratio.

10 CONCLUSIONS

We investigate independent comment changes, commenting guidelines, and tools, to understand how suboptimal comments are introduced and addressed. We find ICCs that aim to improve

suboptimal comments prevalent: ~15.5% of 24M comment changes in 4,392 open-source Java repositories are committed independently of corresponding code, and the ICC ratio tends to decrease as repositories mature. We develop a taxonomy with three dimensions—what kind of comment is changed, how it changed, and what factors are associated with changes—expanding on prior work and adding the *factors behind ICCs*. The identified factors explain the fragility of some comments and the certain commenting tendencies of developers that lead to suboptimal comments. We conduct the study on commenting guidelines, revealing their prevalence, taxonomy, impact (on suboptimal comments), and violations. We also analyze comment checking and generating tools, and reveal the prevalence and ineffective use of tools. We provide insights for conducting research, formulating practices, and improving tools to help facilitate future work.

ACKNOWLEDGMENTS

We thank Milos Gligoric, Haiqiao Gu, Pengcheng Li, Zanpeng Ma, Pengyu Nie, Xin Tan, and Yuxia Zhang for piloting our survey/questionnaire, and Julia Rubin for comments on a paper draft.

REFERENCES

[1] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2018. A Doc2Vec-based assessment of comments and its application to change-prone method analysis. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 643–647.

[2] androidannotations. 2019. How To Contribute Code. https://github.com/androidannotations/androidannotations/wiki/HowToContributeCode.

[3] Apache Maven Javadoc Plugin. 2022. Apache Maven Javadoc Plugin. https://maven.apache.org/plugins/maven-javadoc-plugin/index.html.

[4] Oliver Arafati and Dirk Riehle. 2009. The comment density of open source software code. In *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 195–198.

[5] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python coding style compliance on stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 210–214.

[6] Thomas Ball and Stephen G. Eick. 1996. Software visualization in the large. *Computer* 29, 4 (1996), 33–43.

[7] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[9] bestcommit. Git Commit Best Practices. https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60. (????).

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[11] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. 2019. Automatically detecting the scopes of source code comments. *Journal of Systems and Software* 153 (2019), 45–63.

[12] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[13] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 826–831. https://doi.org/10.1145/3238147.3240471

[14] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. 2003. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*. 77–ff.

[15] commitaksw. 2018. remove auto-generated useless comments. https://github.com/AKSW/RDFUnit/commit/06fe9f13132b3dd74ade48904aa7ff79fa29c751.

[16] commitanki. 2009. Cleaned up old unused code. https://github.com/ankidroid/Anki-Android/commit/b980b1b29a1700f67cea08fa1b57901c2754e8e1.

[17] commitaosp2. 2010. Fix some more typos, remove unused imports, and remove unnecessary cast. https://github.com/aosp-mirror/platform_frameworks_base/commit/bcd573229e5ec6d59bb7931a84a2baa86e0d200a.
[18] commitauthme. 2015. Prepare the project for javadocs. https://github.com/AuthMe/AuthMeReloaded/commit/118c79401a1a7d71102d90aba3db1bfcf4be5211.
[19] commitazure. 2011. Removing "TODO" comments (made them issues in issue tracker). https://github.com/Azure/azure-sdk-for-java/commit/3094bd0ab56395b4a06f98270859c3bbd8c948ba.
[20] commitcommons-math. 2011. Javadoc improvements. Made it more explicit which methods modify instance data. https://github.com/apache/commons-math/commit/9a8435f9d0b0633a76deba573c453d545baff793.
[21] commitgit. 2016. diff: improve positioning of add/delete blocks in diffs. https://github.com/git/git/commit/433860f3d0beb0c6f205290bd16cda413148f098.
[22] commitjavasimon. 2013. code cleaning, removed some more unnecessary javadoc inherits. https://github.com/virgo47/javasimon/commit/c3514caf2574b4f97825b1b3539cffceab8b1588.
[23] commitjikes. 2012. RVM-962: Remove duplicated comments from the packages org.jikesrvm.com. https://github.com/JikesRVM/JikesRVM/commit/1508784d0dabb45b14788ab85187fc394fc78d0f.
[24] commitmodcluster. 2016. JavaDoc cleanup: useless @see clauses, broken links, non-existent. https://github.com/modcluster/mod_cluster/commit/4ab47c63c336732a92686bef5cfbbfc059cc8573.
[25] commitpinpoint. 2014. Translated comments to English. https://github.com/pinpoint-apm/pinpoint/commit/f9715c836612d8b3176ce2b4433fd27e0e890fc2.
[26] commitplatform. 2018. MediaPlayer2: clarify sync/async for APIs. https://github.com/aosp-mirror/platform_frameworks_base/commit/d526bc3b886b4127c310f5dea0035404e28d27a2.
[27] commitqpid. 2015. NO-JIRA: remove stale TODO that has already been taken care of. https://github.com/apache/qpid-jms/commit/eeb59e8478f426e7e8c34211feca9da828ea412e.
[28] commitrichfaces. 2011. reformatted to follow JBoss Community conventions (RF-11019, RF-11020). https://github.com/richfaces/richfaces/commit/c7c155a8e0bc636d19848e7c7a2eeb0bbf7f786d.
[29] commitsnappy. 2011. Fix javadoc. https://github.com/xerial/snappy-java/commit/ebf661dc745bf62639e19b964fe9fdbb5d520c81.
[30] committomcat. 2009. Fix https://issues.apache.org/bugzilla/show_bug.cgi?id=48143. https://github.com/apache/tomcat/commit/c52b1bada15a9f28fccf74364c61e43bec3d1ae7.
[31] committracecompass. 2012. Fix a pile of Javadoc warnings. https://github.com/tracecompass/tracecompass/commit/0283f7ffc576ed1b4e3c80c2614362785fdcdb7f.
[32] committracee. 2014. #16 removes author tags since they add no value and can be inferred. https://github.com/tracee/tracee/commit/566d851e46419c9a1dba6f578f388a624dd51ec1.
[33] commitwro4j. 2014. add a test which proves the problem. https://github.com/alexo/wro4j/commit/9d560ffd31c75221bb99f92f7bdbf063e9b1d885.
[34] D. S. Cruzes and T. Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284.
[35] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. 68–75.
[36] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-aware configuration management for object-oriented programs. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 427–436.
[37] Don A. Dillman, Jolene D. Smyth, and Leah Melani Christian. 2014. *Internet, Phone, Mail, and Mixed-mode Surveys: The Tailored Design Method*. John Wiley & Sons.
[38] Omar Elazhary, Margaret-Anne Storey, Neil Ernst, and Andy Zaidman. 2019. Do as I do, not as I say: Do contribution guidelines match the GitHub contribution process? In *ICSME*.
[39] Derar Eleyan, Abed Othman, and Amna Eleyan. 2020. Enhancing software comments readability using Flesch reading ease score. *Information* 11, 9 (2020), 430.
[40] Dror G. Feitelson. 2021. "We do not appreciate being experimented on": Developer and Researcher Views on the Ethics of Experiments on Open-Source Projects. *CoRR* abs/2112.13217 (2021). arXiv:2112.13217. https://arxiv.org/abs/2112.13217.
[41] Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE'07)*. IEEE, 70–79.
[42] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. 2009. Analyzing the co-evolution of comments and source code. *Software Quality Journal* 17, 4 (2009), 367–394.
[43] Kai Gao, Zhixing Wang, Audris Mockus, and Minghui Zhou. 2022. On the variability of software engineering needs for deep learning: Stages, trends, and application types. *IEEE Transactions on Software Engineering* (2022).

[44] Verena Geist, Michael Moser, Josef Pichler, Stefanie Beyer, and Martin Pinzger. 2020. Leveraging machine learning for software redocumentation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 622–626.
[45] gitdiff. 2022. git-diff - Show changes between commits, commit and working tree, etc. https://git-scm.com/docs/git-diff.
[46] githublan. 2021. Github Language Stats. https://madnight.github.io/githut/#/pull_requests/2021/4.
[47] githutinfo. 2022. GitHut - Programming Languages and GitHub. [EB/OL]. https://githut.info/. Accessed Feb. 2022.
[48] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 213–224.
[49] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing method-level source code histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.
[50] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How good is your comment? A study of comments in Java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 137–146.
[51] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 2019. 9.6 million links in source code comments: Purpose, evolution, and decay. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1211–1221.
[52] Hao He. 2019. Understanding source code comments at large-scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE'19*, Tallinn, Estonia, August 26–30, 2019, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1217–1219. https://doi.org/10.1145/3338906.3342494
[53] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: Prevalence, trends, and rationales. In *ESEC/FSE'21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, August 23–28, 2021, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 478–490. https://doi.org/10.1145/3468264.3468571
[54] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. 200–210.
[55] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.
[56] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Zibin Zheng, and Xiapu Luo. 2020. CommtPst: Deep learning source code for commenting positions prediction. *Journal of Systems and Software* (2020), 110754.
[57] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020. Towards automatically generating block comments for code snippets. *Information and Software Technology* 127 (2020), 106373.
[58] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. 2020. Does your code need comment? *Software: Practice and Experience* 50, 3 (2020), 227–245.
[59] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.
[60] Google Inc. 2020. Google C++ Style Guide. (2020). https://google.github.io/styleguide/cppguide.html.
[61] Jautodoc. 2022. JAutodoc - Eclipse Plugin. http://jautodoc.sourceforge.net/.
[62] Jautodocissue. 2015. Javadoc for public APIs. https://github.com/psi-probe/psi-probe/issues/540.
[63] Zhen Ming Jiang and Ahmed E. Hassan. 2006. Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*. 179–180.
[64] kiegroup/droolsjbpm-build-bootstrap. 2022. README.md. https://github.com/kiegroup/droolsjbpm-build-bootstrap/blob/main/README.md.
[65] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.
[66] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 154–163.
[67] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.
[68] Frederic M. Lord. 1964. The effect of random guessing on test validity. *Educational and Psychological Measurement* 24, 4 (1964), 745–747.

[69] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. 2020. Where should I comment my code? A dataset and model for predicting locations that need comments. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[70] Walid Maalej and Martin P. Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[71] Paul W. McBurney and Collin McMillan. 2016. An empirical study of the textual similarity between source code and source code summaries. *Empirical Software Engineering* 21, 1 (2016), 17–42.

[72] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[73] Andrew Nesbitt and Benjamin Nickolls. 2017. Libraries.io open source repository and dependency metadata. (2017).

[74] Tien N. Nguyen, Ethan V. Munson, and John T. Boyland. 2005. An infrastructure for development of multi-level, object-oriented configuration management services. In *ICSE*.

[75] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. 2019. A framework for writing trigger-action todo comments in executable format. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 385–396.

[76] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: Is it worth it for small programming tasks? *Empirical Software Engineering* 24, 3 (2019), 1418–1457.

[77] oracalguidelines. Code Conventions for the Java TM Programming Language. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html. (????).

[78] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers—Taxonomies and characteristics of comments in operating system code. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 331–341.

[79] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 161–170.

[80] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. 2020. Associating natural language comment and source code entities. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*. 8592–8599.

[81] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *AAAI Conference on Artificial Intelligence*. 427–435.

[82] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[83] Luca Pascarella. 2018. Classifying code comments in Java mobile applications. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 39–40.

[84] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.

[85] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019), 1499–1537.

[86] Fazle Rabbi and Md. Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th International Conference on Program Comprehension*. 371–375.

[87] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 2021. Do comments follow commenting conventions? A case study in Java and Python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 165–169.

[88] Pooja Rani, Mathias Birrer, Sebastiano Panichella, Mohammad Ghafari, and Oscar Nierstrasz. 2021. What do developers discuss about code comments? In *International Working Conference on Source Code Analysis and Manipulation*. to appear. http://arxiv.org/abs/2108.07648.

[89] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of Systems and Software* 181 (2021), 111047.

[90] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Mohammad Ghafari, and Oscar Nierstrasz. 2021. What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk. *Empirical Software Engineering* 26, 6 (2021), 1–49.

[91] Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 112–122.

[92] Sébastien Rufiange and Guy Melançon. 2014. AniMatrix: A matrix-based visualization of software evolution. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 137–146.
[93] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. 2018. Analyzing code comments to boost program comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 325–334.
[94] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of coding standard conformance in open-source data science projects. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.
[95] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 89–92.
[96] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 43–52.
[97] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 83–92.
[98] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 251–260.
[99] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. 2020. Towards detecting inconsistent comments in Java source code automatically. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 65–69.
[100] Supplementary Materials. Supplementary Materials. Website. (????). https://github.com/TyphoonWang/Suboptimal-Comments.
[101] surveysample. Sample Size Formulas for our Sample Size Calculator - Creative Research Systems. https://www.surveysystem.com/sample-size-formula.htm. (????).
[102] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments? */. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. 145–158.
[103] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 11–20.
[104] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.
[105] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An exploratory study of deep learning supply chain. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 86–98.
[106] Xin Tan and Minghui Zhou. 2019. How to communicate when submitting patches: An empirical study of the Linux kernel. *Proc. ACM Hum. Comput. Interact.* 3, CSCW (2019), 108:1–108:26. https://doi.org/10.1145/3359210
[107] Xin Tan, Minghui Zhou, and Zeyu Sun. 2020. A first look at good first issues on GitHub. In *ESEC/FSE'20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event*, USA, November 8–13, 2020, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 398–409. https://doi.org/10.1145/3368089.3409746
[108] Sider Team. 2021. How Open-Sourced Projects use Checkstyle. [EB/OL]. https://siderlabs.com/blog/an-overview-of-checkstyle-and-how-it-is-used-in-open-sourced-projects-8dc288f65fdb/. Accessed Feb. 2021.
[109] Gias Uddin and Martin P. Robillard. 2015. How API documentation fails. *IEEE Software* 32, 4 (2015), 68–75.
[110] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 38–49.
[111] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.
[112] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.
[113] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 334–344.
[114] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *ICSE'20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

[115] Jingyi Zhang, Lei Xu, and Yanhui Li. 2018. Classifying Python code comments based on supervised learning. In *International Conference on Web Information Systems and Applications*. Springer, 39–47.

[116] Yuxia Zhang, Hui Liu, Xin Tan, Minghui Zhou, Zhi Jin, and Jiaxin Zhu. 2022. Turnover of companies in OpenStack: Prevalence and rationale. *ACM Transactions on Software Engineering and Methodology* (2022).

[117] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. 2021. Companies' participation in OSS development-An empirical study of OpenStack. *IEEE Trans. Software Eng.* 47, 10 (2021), 2242–2259. https://doi.org/10.1109/TSE.2019.2946156

[118] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. 2015. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software* 99 (2015), 175–198.

[119] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.

Received 13 December 2021; revised 13 June 2022; accepted 20 June 2022