An Extensive Study of Independent Comment Changes in Java Projects

***Abstract*—While code comments are valuable for software development, code often has low-quality comments or misses comments altogether, which we call *suboptimal comments*. Such suboptimal comments create challenges in code comprehension and maintenance. Despite substantial research on suboptimal comments, empirical knowledge about why comments are suboptimal is lacking, affecting commenting practice and related research. We help bridge this knowledge gap by investigating *independent comment changes*—comment changes committed independently of code changes—which likely attempt to address suboptimal comments. We collect 23M+ comment changes from 4,410 open-source Java repositories and find that ∼16% of comment changes are independent, indicating a considerable amount of comments may be suboptimal. Our thematic analysis of 3,600 randomly sampled independent comment changes provides a two-dimensional taxonomy about what is changed (comment category) and how it changed (commenting activity category). We find some combinations of comment and activity categories have a relatively high frequency although those comments are *not* a large proportion of all comments; the reason may be that some comments easily become obsolete/inconsistent. By further inspecting extensive related materials for these independent comment changes, and validating it with a survey of 33 developer respondents, we find four reasons for suboptimal comments: belief in future actions, lack of comment guidelines, ineffective use of tools, and legacy. We finally provide implications for project maintainers, researchers, and tool designers.**

I. Introduction

Software developers frequently write comments[1] along with source code. Comments are considered an essential form of documentation [20], are present in almost all software systems [15], [26], provide valuable information for program comprehension [56], [60], and are known to be especially helpful for developers to understand code written by others [41]. For example, Google's coding style guide states, "comments are absolutely vital to keep our code readable" [32].

However, comments do not directly impact software functionality, so many comments may not have sufficient quality, may not be properly maintained, or may be missing altogether. In this paper, we use the term *suboptimal comments* to refer to all such cases of comments with low quality (inconsistent, obsolete, useless, missing important information) or even missing altogether. Such suboptimal comments create challenges in code maintenance and reuse [53], [57] but are still prevalent in practice. For example, simple searches of GitHub (in June 2020) return (1) 15K+ issues with "missing Javadoc" in the title, where desired Javadoc comments are missing; and (2) 204K+ issues with "outdated comment" in the title. Figure 1 shows a real example issue where missing Javadoc comments made it difficult to understand some code.

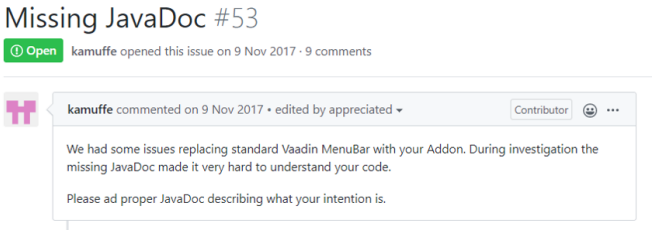


Fig. 1. An example issue from GitHub reporting the problem brought by missing Javadoc comments

Despite prior research on suboptimal comments [17], [29], [30], [35], [36], [40], [47], [52], [53], [54], [55], [63], empirical knowledge about why comments are suboptimal remains sparse. Bridging this knowledge gap can help developers improve comment quality, formulate best practices, and build better tools. However, directly identifying suboptimal comments remains challenging despite recent research progress. There is no unified metric for comment quality [51]. Matching code and comments is difficult in general [18], [59], especially for non-Javadoc comments; it is even harder to check if comments and code are semantically consistent [17], [40], [53], [55], and research on identifying inconsistent comments is limited to specific scenarios, e.g., usage in Javadoc comments [17] or TODO comments in specific format [40]. Research on proposing where to add comments is promising but preliminary [29], [30], [36]. (More details in Section VI).

Our key insight is that we can learn about suboptimal comments by studying *independent comment changes*, which modify only comments but not the code corresponding to the comments. Such independent comment changes likely aim to improve overall comment quality (by adding new comments, updating existing comments or deleting poor comments). We address the following research questions, which would help bridge the knowledge gap about suboptimal comments.

- RQ1: How frequent are independent comment changes? We collect 23M+ comment changes (12M+ Javadoc and 11M+ non-Javadoc comment changes) from commits of 4,410 open-source Java repositories. We find a considerable amount of independent comment changes exist broadly in these repositories. In particular, ∼3.7M comment changes are independent, accounting for ∼16% of all comment changes; independent Javadoc/non-Javadoc comment changes range from 7.0%/7.1% to 22.4%/15.4% (interquartile range) across the repositories.

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

- RQ2: What comments are often changed independently and how are they changed?
  We conduct a thematic analysis on randomly sampled 3,600 independent comment changes. We generate a two-dimensional taxonomy that explains what is changed (we identify four comment categories and 13 subcategories) and how it changed (we identify six commenting activity categories). Some combinations of comment and activity have a relatively high frequency although these comments are *not* a large proportion of all comments, whose reason may be some comments easily become obsolete/inconsistent, or developers overlook certain comments.
- RQ3: Why are comments suboptimal?
  We combine (1) a manual inspection of extensive materials (including commit message, PR information and GitHub issue if any, along with comment change) for independent comment changes and (2) a survey of active developers (with 33 respondents from 400 developers). We summarize four reasons why developers have suboptimal comments, including belief in future actions, lack of comment guidelines, ineffective use of tools, and legacy.

Our results reveal problems in commenting practices and inspire several recommendations on formulating best practices, building better tools, and conducting research (Section IV). Our anonymized data is available at https://drive.google.com/drive/folders/123rA_vV-_9x5jK8-7odzA6aPfJIXINaT.

II. Methodology

To learn about suboptimal comments, we study independent comment changes, which likely aim to improve overall comment quality. We retrieve all comment changes from thousands of open-source repositories (Section II-A), identify independent comment changes for RQ1 (Section II-B), establish a taxonomy of them for RQ2 (Section II-C), and analyze their underlying reasons for RQ3 (Section II-D).

A. Data Preparation

We first select sufficient representative repositories and collect all comment changes from these repositories.

1) Selecting Repositories: We select repositories from GitHub, considering several criteria. Following previous studies [22], [23], [59], we only target Java projects because of Java's maturity and popularity. We want to select repositories with mature practices and rich development history so that the commenting practices we observe would be common, and the obtained lessons could broadly generalize.

We use Libraries.io [39] to obtain the GitHub repository metadata. To exclude personal or toy repositories, we only keep repositories with at least 10 stars, 10 forks, and 5 contributors, following a previous study [59]. We also exclude forked repositories and eventually obtain 8,252 repositories after this step. We further select repositories with more than 500 commits (following previous studies [25], [59]) to obtain sufficient history that includes 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" and "interview". We also exclude repositories that we cannot clone at the start of our study (Dec. 2019). We finally obtain 4,410 repositories.

TABLE I
Statistics of comment-line changes in selected repositories

#Repositories	4,410
#Commits	28,020,418
#File changes of Java files	45,698,099
#File changes of Java files with a comment-line change	17,726,762
#Hunks with a comment-line change	28,980,802
#Added comment-lines	58,069,831
#Deleted comment-lines	46,494,429

2) Collecting Hunks with Comment-Line Changes: After cloning these 4,410 repositories, we use `git` to retrieve all commits in each repository. Each commit may have one or several file changes, and we retrieve all changes of Java files. Each file change has one or more *hunks*, and each hunk may contain multiple added or deleted lines, which we call *changed lines*. For each 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 a script with two simple (but accurate) heuristics. We first match an entire hunk using one regular expression to see if the hunk contains a multi-line comment, which may be a Javadoc comment; if so, we label all the lines that match the regex as being comment-lines. We then match each changed line using another regex to see if the line contains an non-Javadoc comment. To evaluate our heuristic, we manually check randomly sampled 1,000 hunks and find the recall and the precision of our heuristic to be 97.6% and 100%, respectively, for the changed comment-lines. Table I shows the statistics of comment-line changes.

B. Identifying Independent Comment Changes

To identify independent comment changes, we need to decide whether a comment change is accompanied by changed corresponding code. We employ two simple strategies for the two types of comments, Javadoc and non-Javadoc comments.

For Javadoc comments, we consider only comments for methods and classes, because for them we can find the corresponding code. This code could be in *other hunks*, so when any hunk in a changed file contains a Javadoc comment-line, we analyze the entire file at once. For each changed Javadoc comment, we check if any code line in the corresponding method or class changed. If no code line changed, we consider this comment change as an independent Javadoc comment change. The mapping between hunks and Javadoc comments may be *many-to-many*: one hunk may contain changes to zero, one, or multiple Javadoc comments.

For non-Javadoc comments, we only exclude license headers. Based on the finding that about 90% of comments are about nearby code [24], we assume that code changes in the same hunk would correspond to these comments, as `git` applies complex heuristics to ensure one hunk contains *all* nearby changes [7]. Thus, for non-Javadoc comments, we determine whether all the lines in a hunk are comment-lines or not. If yes, we consider the entire hunk as *one* independent

non-Javadoc comment change, even if multiple comment-lines were added or deleted. In general, it would be hard to tell the number of semantically separate comments there were actually changed when a hunk has multiple comment-lines changed.

C. Construction of Taxonomy

With independent comment changes collected, we next categorize these changes to study what comment elements are changed and how. Understanding how independent comment changes aim to improve overall comment quality can help to understand how comments were suboptimal before the changes. Specifically, we conduct a thematic analysis [19] of sampled comment changes following the steps below.

(1) We distinguish six types of comment changes, i.e., added/deleted/updated Javadoc/non-Javadoc comment changes, and randomly sample 600 independent comment changes of each type. In total we selected 3,600 independent comment changes. In this paper, 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).

(2) Three authors independently conduct comment analysis following these steps: (a) Generate initial codes. For each comment change, we carefully read the changed comment, corresponding code, and related commit message, and we generated the initial codes that characterize the nature of changed comments and how they are changed (commenting activity). (b) Group initial codes that have similar key information. All initial codes are organized into themes, suggesting the nature of changed comments and how they are changed. At the end of this step, we reviewed and merged similar themes. (c) Define the final themes. We considered each theme (i.e., category of comment or commenting activity), whether it contains sub-themes (i.e., subcategory), and how these sub-themes interacted and related to the main theme.

(3) For conflicted codes and themes, the three authors would discuss and the final judgement is made by a non-author arbitrator who has more than six years of Java programming experience and conducted similar qualitative analyses before. We had a few comment changes that no author could understand despite our best effort, so we labelled them as *noise*.

Eventually, we acquire a two-dimensional taxonomy of independent comment changes that classifies these changes based on what kind of information is changed (four categories/thirteen subcategories of comment) and how it changed (six categories of commenting activity). During independent labelling, the inter-rater reliability is 0.79 (Cohen's Kappa), which suggests a substantial agreement.

D. Identifying Reasons Behind Suboptimal Comments

We combine a manual inspection and a survey to understand the reasons behind suboptimal comments.

1) Manual inspection: To understand the reasons behind suboptimal comments, we manually inspect sampled 3,600 comment changes and their related materials for all categories. For each comment change, we first collect all related materials through GitHub, including commit messages, related issue reports, and pull request information. Then we read these materials for each category, and try to reveal the pattern of commenting practice and the reason behind the practice. For some categories, we find some patterns on commenting practice, which leads us to the reasons behind suboptimal comments, while for other categories we do not find any pattern. Next, we merge similar patterns and extract the reason behind these patterns. Eventually, we summarize four reasons based on four patterns we discovered.

2) Survey of developers: We validate our four summarized reasons through a survey of developers. We follow the principles of Dillman et al. [21] to design the survey. Based on our previously obtained four reasons, we design a questionnaire that asks if developers witnessed the practices related to the four reasons, whether they consider the practices to negatively impact project maintenance and how they can be improved. We first conduct a pilot study with four researchers with experience 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 comment examples in instructions and eliminated questions that respondents had difficulty answering. The final questionnaire includes three background questions with demographic purpose, 19 choice questions that investigate how developers behave in four scenarios related to the four reasons and how they perceive current commenting practice in these scenarios, and an open-ended question in the end for collecting respondents' additional "insights" and "experience" beyond the scenarios we provide.

To find potential participants with recent software contributions and are familiar with current commenting practices in open-source communities, we selected developers who have contributed more than 10 commits between January 2019 and December 2019. From the repositories we collect, We select 6,059 developers and their email addresses. We randomly sample 400 emails and survey them by sending emails containing our online questionnaire's link. Out of 400 emails, 22 could not be delivered. The survey ran for two weeks, and we received 33 valid responses eventually. The response rate is 8.7%, comparable to the response rate (6% to 36%) in other surveys of software engineering studies [49].

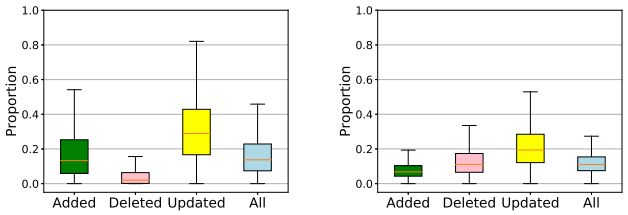
III. Results

A. RQ1: How frequent are independent comment changes?

Table II summarizes the results of our analysis of 4,410 repositories. We find that 15.97% = (1951278+1795212) / (12049563+11406209) of all comment changes are independent. In fact, some comment changes could be committed with irrelevant code changes, so the proportion of actually semantically independent comment changes may be even higher than 15.97%. This high number suggests that considerable amount of comment changes are committed independently, where comments were likely suboptimal before the changes and changes likely aim to improve overall comment quality.

TABLE II
STATISTICS OF THE INDEPENDENT COMMENT CHANGES AMONG ALL COMMENT CHANGES

		Added	Deleted	Updated	All
#Javadoc comment changes	All	6,223,588	3,914,606	1,911,369	12,049,563
	Independent	1,025,688 (16.48%)	193,030 (4.93%)	732,560 (38.33%)	1,951,278 (16.19%)
#non-Javadoc comment changes	All	5,089,020	2,934,565	3,382,624	11,406,209
	Independent	410,496 (8.07%)	438,224 (14.93%)	946,492 (27.99%)	1,795,212 (15.74%)



(a) Javadoc changes (b) non-Javadoc changes

Fig. 2. Proportion of independent comment changes in 4,410 repositories

Across types of comments, 16.19% / 15.74% of Javadoc / non-Javadoc comment changes are independent. Across types of comment changes, 16.48% / 8.07% of Javadoc / non-Javadoc added comment changes are independent, 4.93% / 14.93% of Javadoc / non-Javadoc deleted comment changes are independent, and 38.33% / 27.99% of Javadoc / non-Javadoc updated comment changes are independent.

We find that independent comment changes exist widely in almost all repositories. Specifically, Figure 2 shows the distribution of the proportion of independent comment changes to all comment changes in each repository. Across all 4,410 repositories, the median proportions are about 13% / 6%, 2% / 11%, and 29% / 19% for independent Javadoc / non-Javadoc comment changes that are added, deleted or updated, respectively. Moreover, the proportion of independent Javadoc / non-Javadoc comment changes ranges from 7.0% / 7.1% (lower quartile) to 22.4% / 15.4% (upper quartile). Independent comment changes exist widely: only 166 / 66 of 4,410 repositories have 0% independent Javadoc/non-Javadoc comment changes, while only 25 (0.5%) repositories have no independent comment changes at all.

We conclude with the following finding:

> Finding 1: About 16.0% of 23M+ comment changes in the 4,410 studied repositories are independent, suggesting a considerable amount of suboptimal comments exist before the changes. Across these repositories, the proportion of independent Javadoc / non-Javadoc comment changes ranges from 7.0% / 7.1% to 22.4% / 15.4% (lower to higher quartile), indicating the prevalence of this phenomenon.

B. RQ2: What comments are often changed independently and how are they changed?

As explained in Section II-C, we conducted a thematic analysis of 3,600 independent comment changes. Eventually, we discovered a two-dimensional taxonomy of independent comment changes, with one dimension for categories of comments and the other for categories of commenting activities, as shown in Table III. The table has four categories (with 13 subcategories) of comments that are changed and six categories of commenting activities for the changes. In each cell, the number before '/' is for Javadoc and the number after for non-Javadoc.

The four categories of comments are the following:

- **Code Logic**: Comments that describe the code behavior; subcategories: **Functionality summary**: Comments that summarize the code functionality, including the functionality of certain code fragments and the explanation for variables. **Context**: 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. **Rationale**: Comments that explain why the corresponding code fragment is used or why a certain algorithm is applied. **Code file structure**: 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; subcategories: **TODO**: Comments that document unfinished work or unfixed bugs. **Commented code**: Commented source code, including code that was used for testing/debugging and code examples in Javadoc. **Empty or Uninformative**: 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; subcategories: **Auto-generated**: Comments generated by tools or IDE plugins. **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; subcategories: **Log**: Comments that document the author and version information. **Link**: Links in comments, including `@see` and `@link` tags in Javadoc and URL links.

Our comment taxonomy is inspired by the taxonomy of Java comments proposed by Pascarella and Bacchelli [45], but we make some important revisions, extending three of their subcategories, merging two subcategories, and reorganizing categories. In particular, our manual inspection finds three kinds of comments that cannot fit in their taxonomy: (1) comments explaining code structure, so we name the new

TABLE III
NUMBER OF JAVADOC / NON-JAVADOC COMMENT CHANGES PER CATEGORY. THE TOTAL SUM IS 3,878, GREATER THAN THE NUMBER OF INSPECTED COMMENT CHANGES BECAUSE ONE CHANGE MAY INVOLVE MULTIPLE CATEGORIES, E.G., A NEWLY ADDED JAVADOC MAY CONTAIN BOTH FUNCTIONALITY SUMMARY AND USAGE.

		∼Freq. in [45]	Adding Entirely New Comment	Deleting Obsolete Comment	Adding Supplementary Info	Fixing Inconsistency	Clarifying Description	Formatting and Others
Code Logic	Functionality Summary	40.5%	328 / 137	109 / 27	43 / 16	11 / 39	60 / 27	Translation: 12 / 12
	Context	1.5%	33 / 73	8 / 30	24 / 24	8 / 68	6 / 22	
	Usage	23.9%	204 / 0	95 / 0	54 / 0	30 / 0	27 / 0	Adjusting lines and spaces: 300 / 157
	Rationale	1.9%	9 / 47	2 / 3	0 / 2	0 / 4	0 / 4	
	Code File Structure	0.4%	0 / 31	0 / 29	na / na	0 / 8	0 / 1	
Under Development	TODO	1.4%	7 / 144	7 / 140	0 / 36	1 / 0	0 / 21	Updating tags: 0 / 6
	Commented Code	2.5%	27 / 118	6 / 286	5 / 0	18 / 0	na / na	
	Empty/Uninformative	0.7%	28 / 3	106 / 10	na / na	2 / 0	na / na	Fixing typo: 34 / 93
Tool Related	Auto-Generated	1.6%	21 / 2	37 / 7	1 / 0	0 / 1	na / na	
	Deprecation (Javadoc)	0.4%	14 / 0	19 / 0	na / na	na / na	na / na	Moving comments: 0 / 5
	Directive	7.8%	10 / 56	44 / 21	na / na	0 / 1	na / na	
Metadata	Log	4.3%	30 / 7	83 / 17	11 / 0	9 / 4	na / na	Noise: 0 / 3
	Link	13.1%	54 / 12	47 / 42	9 / 0	68 / 16	na / na	
Sum		100%	**765 / 630**	**563 / 612**	**147 / 78**	**147 / 141**	**93 / 75**	**346 / 276**

subcategory *Code File Structure*, extending their subcategory *Formatter*; (2) comments with URL links are not covered by their subcategory *Pointer*, so we establish subcategory *Link*; (3) comments that document the version information or the time when code was introduced, so we establish subcategory *Log*, extending their subcategory *Ownership*. We also merge their subcategories *Exception* and *Usage* because the information on exceptions is often tangled with usage. We reorganize some of their categories to better accommodate added/merged subcategories and to use more accurate names. For example, we add all subcategories related to code logic together and use *Code Logic* instead of their *Purpose*. For convenience of comparison between comment frequencies in changes and code files, we add the approximate ratio from [45] in a column of Table III. For the new subcategories we propose, we estimate the ratio using frequencies of subcategories in [45] with a similar but narrower definition.

The six categories of commenting activities are these:

- **Adding New Comment**: Introducing a new comment into code files.
- **Deleting Obsolete Comment**: Deleting a comment.
- **Adding Supplementary Information**: Adding more information to existing comments.
- **Fixing Inconsistency**: Fixing an inconsistency between code and comment.
- **Clarifying Description**: Updating comments to just restate the expression without introducing or deleting any information.
- **Formatting and Others**: Changing the format of comments or performing other minor activities.

To the best of our knowledge, ours is the first taxonomy for commenting activities.

We can observe from Table III that, some combinations of comment and activity categories have higher frequencies than others. Some are higher simply because these kinds of comments are a large proportion of *all comments*, e.g., functionality summary and usage [45]. However, some other categories of comment changes also have a relatively high frequency, although these comments are *not* a large proportion of all comments [45]. (1) Some comments have a much higher frequency of deleting obsolete comments than adding entirely new comments, e.g., commented code, empty or uninformative, auto-generated, and log. These comments may be fragile and easily become obsolete. (2) Some categories have a relatively higher frequency of fixing inconsistency, e.g., link and usage in Javadoc, and context in non-Javadoc. These comments may easily become inconsistent with code. (3) Though context and usage have a much lower ratio than functionality summary in code files [45], they have a similar frequency of adding supplementary info as in functionality summary. The comparison may suggest that developers often overlook context and usage when they add new comments, making context and usage more likely to be missing or insufficient in existing comments.

We conclude with the following finding:

> Finding 2: We discover a two-dimensional taxonomy with four categories (13 subcategories) of comments and six categories of commenting activities, explaining what comments and how change independently. Some combinations of comment and activity have a relatively high frequency although the comments are *not* a large proportion of all comments; the reason may be that some comments easily become obsolete/inconsistent, or developers overlook certain comments when coding.

C. RQ3: Why are comments suboptimal?

As explained in Section II-D, by combining a manual inspection of comment-change-related materials and a survey with 33 developer respondents, we obtain four reasons for suboptimal comments: belief in future actions, lack of comment guidelines, ineffective use of tools, and legacy.

1) Belief in future actions: One reason for suboptimal comments is that developers believe that some comments are temporary and will be changed in the future. We observe two common scenarios. In one scenario, developers add comment skeletons, believing they will be completed later, but that rarely happens. In the other scenario, developers comment out code, believing they may uncomment or delete it later, but actually do not.

a) Adding Skeleton: In our manual inspection, we discover that developers add empty or uninformative comments, but rarely complete and often delete them. Our analysis of the commits for the 31 added empty or uninformative comments (28 / 3 in Table III) finds that developers often leave messages about completing these skeletons in the future. For example, in the commit `2ed2dea1` [11] in `lieray/liferay-portal`, the commit message says "*Jim knows to document this class. I left the empty javadocs in there*", and the empty Javadoc is indeed completed by another developer later [6].

However, not all empty or uninformative comments are completed eventually, and most are deleted uncompleted. For example, we investigate how all newly added 28 empty or uninformative Javadocs in Table III evolved. Of those 28, only 5 (17.9%) have been supplemented with more information, while 23 (82.1%) still remain uninformative or were deleted uncompleted. Table III also shows that 106 empty or uninformative Javadocs were deleted in our sample, which account for 18.8% of 563 deleted obsolete Javadocs, again showing that many skeletons are just deleted rather than completed.

Leaving empty or uninformative comments appears to be a common practice in open source. We run a script to identify empty or skeleton Javadocs in the version of 4,410 Java repositories retrieved for our study. We find that 195,525 Javadocs are empty, and 195,759 Javadocs are skeletons only with empty Javadoc tags. These ∼400k empty or skeleton Javadocs account for ∼2.5% of all 16M existing Javadocs.

b) Commented Code: Our manual inspection confirms the common pattern that developers tend to first comment the unused code snippets instead of deleting them directly, which matches the finding from Pham and Yang [46] that the primary motivation of commenting code is to mark it for later removal. However, developers rarely maintain commented code timely, and commented code is often deleted in batch.

We find considerable deletion of commented code. Our sample contains 286 deleted commented code. The developers deleted this commented code likely because they consider it useless, e.g., in the commit `8fba556e` [3] in `bardsoftware/ganttproject`, the commit message says "*[r]emoved meaningless comments and commented code*", and the comment deleted commented code in several files.

c) Validation via survey: 63.6% of respondents report that they have performed themselves or witnessed others leaving comments that need further maintenance. Moreover, 52.4% of respondents report that developers add these comments to mark what should be updated in the future, while 28.6% report that they added these comments without any specific purpose, and 19.1% report that they expect others with expertise to complete the comments. Respondents perceive that most of these comments are not managed in the future: 52.4% of respondents think Javadoc skeletons are most often ignored and remain incomplete, 42.9% think commented code is most often eventually deleted, and 33.3% think it is most often ignored and remains commented. Moreover, most respondents (57.6%) believe that introducing these comments negatively impacts project maintenance. Respondents also suggest how to improve current practice (more details in Section IV-A).

We conclude with the following finding:

> Finding 3.1: One reason for suboptimal comments is that developers believe some comments such as skeletons they add or code they comment out will be managed in the future, but the followup is rarely taken.

2) Lack of comment guidelines: Some open-source communities have their own guidelines for code conventions while many others do not, and even those that have guidelines for code conventions may not include guidelines for comment writing. Some communities do have guidelines for comment writing, e.g., the style guide [4] in `Android` requires that "*[y]ou must add @params and @returns for each parameter/return value*". In those open-source communities without such guidelines, especially in small repositories, developers may hold different ideas on writing comments, which may lead to the missing or confusing comments.

We discover that two aspects, *what* to write and *how* to write, often lead to suboptimal comments when missing in guidelines.

a) What to write: When inspecting related materials of comment changes that add entirely new comments or supplementary info for code logic, we find the pattern that developers do not add comments with code timely because they do not realize which code deserves commenting. In a repository without guidelines on what to write, developers may miss some significant information during commenting.

Context and usage information are especially easily overlooked by developers. Context and usage have a much lower frequency than functionality summary in code files [45] and in the adding entirely new comment column in Table III, while they have an similar frequency in the column adding supplementary info. The comparison suggests that context and usage are more likely to be overlooked by developers than functionality summary when adding new comments. For example, the commit `34d92ef6` in repository `aosp-mirror/platform_frameworks_base` states "*[a]dded @throws tag in the javadoc for many APIs which might throw a SecurityException in cases when such information might be useful for the caller*", where the exception information was overlooked before the change. If commenting guidelines stipulate what to write, e.g., which APIs require usage in Javadoc, the issue could be alleviated.

b) How to write: Without guidelines on how to write comments, developers may use inconsistent terms or write

imprecise descriptions that require future updates. Moreover, the format of comments may be inconsistent in the repository.

In our manual inspection, we find the pattern that some changes do not modify the information in the comments but just rephrase the description to make it easier to be understood by others or to make terminology consistent with other comments. For example, the issue `#2789` [1] in repository `realm/realm-java` raised the problem of inconsistent terms in Javadoc and website documentation, and it is fixed in pull request `#2828` [14] that states "*[t]his PR changes the terminology on the Java side so object not in Realm are called 'unmanaged' everywhere*". In Table III, 93 / 54 Javadoc / non-Javadoc independent comment changes about code logic just clarify description.

Moreover, in the last column of Table III, 24 comment changes are translation and 457 comment changes just adjust newlines or blank lines. These comment changes attempt to unify the language and format of comments to improve the readability of comments. The inconsistencies of language and format may be brought by the lack of guidelines on how to write comments.

c) Validation via survey: According to the survey, many open-source communities do not have guidelines on commenting. 27.3% of respondents reported that the projects they have participated in never had guidelines on commenting, and 42.4% reported that most of them had no such guidelines. The survey also validates the negative impact of lack of comment guidelines: 51.5% of respondents think that lack of comment guidelines would negatively impact comment quality and code maintenance. 78.8% of respondents think that commenting guidelines should include what to document in comments (what to write), while 54.5% and 48.5% of respondents think the guidelines should also include the format and language style (how to write), respectively. Respondents also provide open opinions in "other" option. For example, a few respondents comment on the guidelines and general principles: three respondents think comments should be minimal and only document the "non-obvious", one states comment writers should not *"assume others can read your mind"*. One respondent expresses concern on the enforcement of guidelines, *"...do not see a better way except for peer review"*.

We conclude with the following finding:

> Finding 3.2: Lack of comment guidelines, especially on what to write and how to write may lead to missing, inconsistent, or imprecise comments. Developers write imprecise descriptions or use inconsistent terms, format and language in comments, sometimes refining these comments later.

3) Ineffective Use of Tools: We find that the ineffective use of tools leads to suboptimal comments. In particular, comment auto-generating tools may lead to uninformative comments that do not explain code, and the inconsistencies in links and annotations could be mitigated by using comment-checking tools like Javadoc and Checkstyle but are often overlooked.

a) Comment Auto-generating Tools: In our manual inspection, we find that comments generated by comment auto-generating tools can often be deleted as useless comments. The pattern suggests that some use of comment auto-generating tools may add uninformative comments.

In Table III, 21 newly added Javadocs and three non-Javadoc comments in our sample appear to be generated by automatic tools or IDE plugins, based on the commit message. For example, the message of the commit `fbccc3e4` [8] in `comunes/kune` says "*JAutodoc code format*", indicating that developers used JAutodoc, an Eclipse plugin for adding Javadoc automatically. Meanwhile, 37 auto-generated Javadocs and seven non-Javadoc comments are deleted due to being considered useless, based on the commit messages for these comment changes. For example, in the commit `06fe9f13` in `AKSW/RDFUnit`, the commit message says "*remove auto-generated useless comments*", and the commit deletes generated comments in 203 files. According to our manual inspection, deleted auto-generated Javadocs contain the following two types: (1) **Templates** generated automatically that only provide unimportant information, e.g., code authorship and committed time. (2) **Useless Javadocs** that only restate the method information already present in the signature.

b) Comment-Checking Tools: In our manual inspection for comment changes that fix inconsistencies in usage and link, we discover the pattern that developers tend to fix these inconsistencies in batch following reported warnings and errors from comment-checking tools like Javadoc and Checkstyle, instead of fixing them timely in the same commit when the code is changed. The pattern suggests that comment-checking tools are often overlooked by developers, leaving comments suboptimal.

Similarly, the inspection of comment changes that fix inconsistencies in usage and link finds that some inconsistencies can be detected by related tools like Javadoc and CheckStyle, including inconsistent `@param` and `@return`, and broken `@see` and `@link`. From our inspection of commit messages and related code, we find these tools can detect 13 deleted and 6 updated inconsistent usage tags in Javadoc, and 17 deleted and 43 updated broken `@see` and `@links`. The warnings and errors raised by the tools are not fixed in the commit where the inconsistency was introduced; instead, developers tend to fix them in batch. Of the aforementioned 79 fixes, 64% belong to large commits containing many comment fixes (more than 10) to eliminate warnings from comment-checking tools. For example, in the commit`8e2abf97` [2] in `apache/ambari`, inconsistencies between Javadoc and code in 46 files are fixed, including unresolved references and wrong parameters.

c) Validation via survey: The survey responses show that developers often use comment auto-generating tools and comment-checking tools. 60.6% of respondents report that they used or witnessed the use of comment auto-generating tools, and the proportion for comment-checking tools is 78.8%. Only 10% of respondents think comment auto-generating tools can generate good comments, while 65% think they cannot. 34.6% of respondents think developers *often* ignore warnings

(or even errors) reported by comments checking tool, while 7.8% think they *always* ignore, 19.2% think they *never* ignore and 19.2% think they *rarely* ignore. 81.0% of respondents think developers ignore these warnings because they believe most warnings are not critical, and 33.3% think developers tend to accumulate minor issues and fix them together.

We conclude with the following finding:

> Finding 3.3: The ineffective use of comment auto-generating tools and comment-checking tools leaves some suboptimal comments. Most comment auto-generating tools only generate uninformative templates. Comment-checking tools can detect some cases where the comment is inconsistent, but developers often overlook their warnings and do not fix broken comments timely. Developers ignore them considering them not critical, and tend to accumulate many warnings to fix them together.

4) Legacy: Log information, i.e., code authorship and version information, is recorded in some comments. However, in our manual inspection, developers tend to delete log information. Some open-source repositories decide to stop writing log information in comments for two reasons.

One reason is that it is hard to maintain, e.g., the author tag in Javadoc, because multiple developers may touch the corresponding method in an open-source community. An issue [10] for `RedHat` says that "[t]he author tags in the java files are a maintenance nightmare". It also complains that a large percentage of author tags are wrong, incomplete, or inaccurate. Thus, it suggests to strip all author tags from Java files.

The other reason is that its function can be replaced by version-control systems like Git. Most of the change history and code authorship can be obtained from version-control systems, so it is unnecessary to document and maintain them in comments. In the commit`65ee0b5f` [13] in `Jasig/uPortal`, author tags in 1,400 files are removed and the commit message says "[a]uthor tags are not as accurate as version control history for knowing who authored a piece of code."

a) Validation via survey: The survey shows that developers think log information in comments is hard to maintain and can be replaced by version-control systems. 55.6% of respondents believe that log information is hard to maintain as the project evolves, while only 25.9% of respondents believe it is unlikely. As for the question *"[c]an the role of log information in comments be replaced by version-control systems such as Git"*, 54.6% vote for "very likely", 18.2% vote for "likely", while 12.1% vote for "unlikely" and nobody chooses "very unlikely". As for the solution, 69.7% of respondents think stopping using log information in comments and deleting obsolete comments would help, which is already applied by some communities in our observation. 33.3% of respondents think open-source communities should set specifications on the use of log information in comments, and 27.2% of respondents think a tool to detect and revise wrong or incomplete log information may help. One respondent reports that s/he does not think it is a problem, and one reports that s/he thinks `@since` is much more important than `@author`, "since it conveys information about how the capabilities of the software have changed".

We conclude with the following finding:

> Finding 3.4: Log information added in comments often becomes legacy over time and gets deleted eventually because it is hard to maintain, and version-control systems can play the same role.

IV. Implications

Based on our findings, we discuss implications for project maintainers, tool designers, and researchers.

A. Project maintainers

(1) Formulating commenting guidelines: Our study confirms the lack of commenting guidelines in open-source communities and its negative impact on maintenance. Therefore, we recommend communities to formulate their own commenting guidelines with both *general principles* and *actionable rules*, and enforce them in code review or via comment-checking tools. For general principles, we advise core developers to suggest what should and should not be commented, in a way that benefits the project the most, where the specific philosophy might differ (e.g., only document the "non-obvious" for internal modules, but detailed functionality summary and usage information for every public method). For actionable rules, we advise the following: add due date in TODO (Section III-C1), do not commit incomplete skeleton (Section III-C2 and III-C3a), use consistent terms(Section III-C2), no `@author` tags(Section III-C4), use English (Section III-C2), and keep a consistent format (Section III-C2).

(2) Enhancing comment maintenance: Our study finds that comments that need further maintenance are often ignored, despite the developers' belief that they (or others) may address these comments and that developers find it harmful not to address the comments. Therefore, we recommend developers to also use `TODO` mark for incomplete comments and commented code, and use `NOTE` mark for easily outdated comments (e.g., links [25]). Developers should also regularly check and maintain these comments manually or with automated tools. Recently, TrigIt [40] offers a promising approach in this direction.

(3) Setting up CI pipelines: Our survey reveals that developers perceive warnings generated by comment-checking tools as not critical. However, we have identified a set of comment smells (Section III-C3b) that are causing comment maintenance problems, and the existing tools can check them (e.g., Javadoc or Checkstyle). In particular, broken links in `@see` and `@link` tags, and inconsistent `@param` and `@return` can be detected by the existing tools. Given that these tools are often highly configurable, we advise developers to use the

tools to specifically check these smells and even integrate the checking of these problematic smells into CI/CD pipeline.

B. Tool designers

(1) Improving comment auto-generating tools: Our study finds that developers are not satisfied with current comment-generating tools. Some generated Javadocs are deleted because they only restate words in the method signature (Section III-C3a). These generated comments can be considered trivial if they have a high coherence coefficient with code [51], because they provide no additional information. Tool designers may consider filtering those trivial comments before the tool generates comments. The reason behind the trivial comments is that current comment auto-generating tools only consider textual information, while structural information could also be considered. For example, Hu et al. [28] extract both lexical and syntactic information Abstract Syntax Trees (AST) in comment generation.

(2) Improving comment-checking tools: Our results reveal that developers often ignore warnings or even errors reported by comment-checking tools, where the comment is wrong or broken. According to our survey responses, developers ignore these warnings because they think most of them are not critical (e.g., "[t]hey are often considered too minor to even care."). To address this problem, we suggest developers stress these warnings related to wrong or broken comments in a conspicuous way, and distinguish these critical warning from other warnings. The tools could also offer suggestions for fixing the problematic comments, e.g., as Spotless [5] does for formatting issues.

(3) Improving version-control system (VCS): Our study finds that log information is often deleted in comments because it is hard to maintain and replaceable by VCS. For example, most developers argue against using `@author` [9], [48] and documenting change history in comment [12]. However, the most widely used VCS, `git`, cannot fully replace or automatically maintain `@version` and `@since` tags, despite their potential usefulness (Section III-C4). `git` also lacks structural understanding of code. For example, `git blame` only tracks last modification by line, but cannot tell when a code entity (e.g. method) is introduced and by whom. Thus we encourage tool designers to improve VCS to fully replace the use of log information in comments.

C. Researchers

(1) Predicting where to comment: We find that some comments are added independently because the corresponding code has been already introduced and requires further explanation. If we can predict where necessary comment is need in the code file, the problem of missing comments can be alleviated. Previous work [36], [29], [30] attempted to address this problem. They used deep learning models to predict the position of comments in code files and checked if the output matched existing comments. We recommend more work towards this direction, and also suggest that the comment categories can be considered in predicting where to comment, so developers can better understand why and what to comment.

(2) Detecting useless comment: We spot that some comments are deleted because developers think they are useless in our manual inspection. We check the commits with keyword "useless" in commit message within our dataset batch, and find these useless comments are uninformative or just restate the obvious. Deleting useless comments is indeed a practice developers would take in development. However, related research on current commenting practice on useless comments is missing. What comments are considered useless and why they become useless is still unclear. This observation can motivate researchers to study commenting practice on useless comments. Developing automated tools to detect useless comments would also be desirable.

(3) Comment generation: Our study finds that tools are used to generate comments automatically by developers but they are not satisfactory. Through manual inspection, we find that all comment auto-generating tools (1) only target Javadoc comments and (2) only generate functionality summary or irrelevant log information. Related work [16], [27], [28], [34], [38], [58] on comment generation mostly target the method/function granularity as well. Therefore, we encourage researchers to explore whether current models can work well on a smaller granularity and to design models that can generate inline comments for code snippets. The research literature also mostly concerns generating functionality summary [16], [27], [28], [34], [38], [50]. However, the taxonomy of the changed comments we establish indicates that usage information is often added and supplemented. Therefore, we encourage researchers to study how to generate usage information for important methods, i.e., generating explanations of parameters and return value.

V. Threats to Validity

Internal Validity: The independent comment changes we extract from hunks using our method may still be related to some code changes far way from the hunk or even changes in another code file. However, accurate matching of comment change to code change is still an open problem, and there is no reliable way to detect whether a comment change is related to a distant code change. Therefore, we decide to only consider nearby code in our study, and adopt matching heuristics that align with current best practices, e.g. Javadoc of a method concerns the method body.

The definition of our taxonomy may contain ambiguities, and the manual labeling is also prone to errors and conflicts. To mitigate such threat, three labelers and one arbitrator are included in this study, all with rich Java development experiences. Furthermore, we iteratively discussed and refined taxonomy definition and labeling criteria through several pilot studies to ensure agreement and reproducibility in the final taxonomy and labeling process.

We cannot guarantee completeness for the four identified reasons behind suboptimal comments, because our manual inspection reveals that many commits, issues, and PRs only

have incomplete or limited information that we cannot infer reasons from them. Although we validate the identified reasons through a survey, the survey also indicates other factors that might lead to suboptimal comments, e.g., *"No time or energy for filling comments saying the obvious"* as stated by one of our survey respondents. We leave further investigations into this problem for future work.

External Validity: Our study is based on a set of popular Java open-source projects from GitHub, which might be unrepresentative of all Java open-source projects. To mitigate this threat, we adopt selection criteria from previous studies [59], [25], which can select projects with diverse backgrounds, 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 of other programming languages, such as Python, C and JavaScript, may have different commenting practices. However, the implications still appear to be helpful for practitioners in proprietary Java projects and in projects using other programming languages.

VI. Related Work

The most related work to our study fits in two groups, comment classification and comment code co-evolution.

A. Comment Classification

A variety of studies explore the nature of comments and establish taxonomy for the explored comments. Padioleau et al. [42] analyzed and proposed a taxonomy for *comments in operating system code*, to find guidance for developing tools for checking comments. Haouari et al. [24] defined a taxonomy of *comment content and relevance* for investigation of developers' commenting habits in Java. Maalej and Robillard [37] defined a taxonomy of knowledge patterns in *API reference documentation* by inspecting Javadoc comments sampled from JDK. Pascarella and Bacchelli [45], [44] proposed a detailed taxonomy of Java *comment* by manually classifying comments sampled from six OSS repositories. Zhai et al. [61] 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. To better characterize and understand *independent comment changes*, we propose a two-dimensional taxonomy in which the *comment* dimension is derived from [45], and the *commenting activity* dimension is novel, not considered in any of the above studies.

The most related work to ours is Wen et al. [59], which explored how different types of code changes trigger comment changes, and proposed a taxonomy of code-comment inconsistencies fixed by developers by analyzing 500 commits. Different from their *commit-level* classification on inconsistency-fixing commits, our taxonomy is derived from *hunk-level* classification on possible independent changes. Consequently, our taxonomy provides insights not only on fixing code-comment inconsistencies, but also on comment maintenance and quality assurance in general.

B. Code-comment co-evolution

The research on code-comment co-evolution focuses on three aspects: to what extent comments evolve with code [22], [23], [31], [33], how to detect outdated or inconsistent comments [35], [40], [47], [52], [53], [54], [55], [63], and how to update comments automatically [43], [62].

Fluri et al. [22], [23] found from seven Java open-source projects that 3∼10% of the comment changes did not occur in the same revision as the associated code changes. Meanwhile, we find that the average ratio of independent comment changes, *with no associated code changes in the same commit*, is about 16% in 4,410 Java open-source repositories, indicating the omnipresence of suboptimal comments. Our results can better represent the current situation in open-source development.

Tan et al. [53] proposed iComment, a system to extract synchronization-related implicit rules in comments, which represent the meaning of comments, and check if they match the logic of code within the same method. Their work is followed by @tComment [55], which checks whether description of parameters and exception matches corresponding description in Javadoc comments. Nie et al. [40] proposed a framework that can detect obsolete todo comments written in a specific format when corresponding task is completed. Our study complements empirical knowledge on suboptimal comments, which may help related research better understand the background of code-comment inconsistencies.

Liu et al. [62] proposed an approach to automatically update comments based on the corresponding code change and old comment, using a seq2seq model learned from a large number of code-comment co-changes. We believe our work can inspire more work in this direction, by complementing empirical knowledge about how suboptimal comments are really improved in practice.

VII. Conclusions

In this study we investigate independent comment changes to understand the nature of suboptimal comments. We find that 16% of 23M comment changes in 4,410 open-source Java repositories are committed independently of corresponding code, indicating considerable amount of comments may be suboptimal. We develop a taxonomy through manual inspection with two dimensions: what kind of information is changed and how it changed. The fragility of some comments and certain commenting tendency of developers may be the reason for the frequency of comment and activity categories. Combining our manual analysis with a developer survey, we discover four reasons for suboptimal comments: belief in future actions, lack of comment guidelines, ineffective use of tools, and legacy. We provide insights for project maintainers, tool designers, and researchers, and expect that our results help facilitate commenting maintenance, and related tools and research.

References

[1] "Align un-managed/standalone/un-persisted terminology · issue 2789 · realm/realm-java," https://github.com/realm/realm-java/issues/2789.

[2] "Ambari-19149. code cleanup: unresolved references in javadoc · apache/ambari@8e2abf9," https://github.com/apache/ambari/commit/8e2abf97bd2f10bad978d176a53bffd0ad2c7b67.

[3] "Cleanup: · bardsoftware/ganttproject@8fba55," https://github.com/bardsoftware/ganttproject/commit/8fba556e6c184e9e4efdf1e6d94d01d731add0a9.

[4] "Code style guide | android open source project," https://source.android.com/devices/architecture/hidl/code-style#comments.

[5] "diffplug/spotless: Keep your code spotless," https://github.com/diffplug/spotless.

[6] "Edit trash handler javadoc · liferay/liferay-portal@50dfc68," https://github.com/liferay/liferay-portal/commit/50dfc689c56cd374a127b8d43232e5f58912d81e#diff-40eb2a71dce39623893b24790df5f76d.

[7] "git-diff - show changes between commits, commit and working tree, etc," https://git-scm.com/docs/git-diff.

[8] "Jautodoc ; code format · comunes/kune@fbccc3e," https://github.com/comunes/kune/commit/fbccc3e47971e446b3261782a839d56886e1cb2d.

[9] "Javadoc @author tag good practices," https://stackoverflow.com/questions/17269843/javadoc-author-tag-good-practices.

[10] "[jbrules-2895] strip all author tags from java files - red hat issue tracker," https://issues.redhat.com/browse/JBRULES-2895?_sscc=t.

[11] "Jim knows to document this class. i left the empty javadocs in there … · liferay/liferay-portal@2ed2dea," https://github.com/liferay/liferay-portal/commit/2ed2dea10ca32f0319561173f45735047ce2ed92.

[12] "Modification history in a file," https://stackoverflow.com/questions/638912/modification-history-in-a-file.

[13] "Nojira: Remove author javadoc tags · jasig/uportal@65ee0b5," https://github.com/Jasig/uPortal/commit/65ee0b5f4a186ad38d6330b0108130d8e316a629.

[14] "Objects not in realm are now called unmananged everywhere. by cmelchior · pull request 2828 · realm/realm-java," https://github.com/realm/realm-java/pull/2828.

[15] O. Arafati and D. Riehle, "The comment density of open source software code," in *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 2009, pp. 195–198.

[16] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.

[17] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[18] H. Chen, Y. Huang, Z. Liu, X. Chen, F. Zhou, and X. Luo, "Automatically detecting the scopes of source code comments," *Journal of Systems and Software*, vol. 153, pp. 45–63, 2019.

[19] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.

[20] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[21] D. A. Dillman, J. D. Smyth, and L. M. Christian, *Internet, phone, mail, and mixed-mode surveys: the tailored design method*. John Wiley & Sons, 2014.

[22] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[23] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[24] D. Haouari, H. Sahraoui, and P. Langlais, "How good is your comment? a study of comments in java programs," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 137–146.

[25] H. Hata, C. Treude, R. G. Kula, and T. Ishio, "9.6 million links in source code comments: purpose, evolution, and decay," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1211–1221.

[26] H. He, "Understanding source code comments at large-scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1217–1219.

[27] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 200–210.

[28] ——, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[29] Y. Huang, X. Hu, N. Jia, X. Chen, Z. Zheng, and X. Luo, "Commtpst: Deep learning source code for commenting positions prediction," *Journal of Systems and Software*, p. 110754, 2020.

[30] Y. Huang, N. Jia, J. Shu, X. Hu, X. Chen, and Q. Zhou, "Does your code need comment?" *Software: Practice and Experience*, vol. 50, no. 3, pp. 227–245, 2020.

[31] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[32] G. Inc. (2020) Google c++ style guide. [Online]. Available: https://google.github.io/styleguide/cppguide.html

[33] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in PostgreSQL," in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 179–180.

[34] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[35] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 154–163.

[36] A. Louis, S. K. Dash, E. T. Barr, M. D. Ernst, and C. Sutton, "Where should i comment my code? a dataset and model for predicting locations that need comments," in *2020 IEEE/ACM 42st International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[37] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[38] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.

[39] A. Nesbitt and B. Nickolls, "Libraries. io open source repository and dependency metadata," 2017.

[40] P. Nie, R. Rai, J. J. Li, S. Khurshid, R. J. Mooney, and M. Gligoric, "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*, 2019, pp. 385–396.

[41] S. Nielebock, D. Krolikowski, J. Krüger, T. Leich, and F. Ortmeier, "Commenting source code: is it worth it for small programming tasks?" *Empirical Software Engineering*, vol. 24, no. 3, pp. 1418–1457, 2019.

[42] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers—taxonomies and characteristics of comments in operating system code," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 331–341.

[43] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," *arXiv preprint arXiv:2004.12169*, 2020.

[44] L. Pascarella, "Classifying code comments in java mobile applications," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2018, pp. 39–40.

[45] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 227–237.

[46] T. M. T. Pham and J. Yang, "The secret life of commented-out source code," in *ICPC*, 2020.

[47] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 112–122.

[48] V. Ruzicka, "Stop using javadoc @author tag," https://www.vojtechruzicka.com/stop-using-javadoc-author-tag/.

[49] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *2013 6th In-*

ternational Workshop on Cooperative and Human Aspects of Software Engineering (CHASE). IEEE, 2013, pp. 89–92.

[50] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[51] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 83–92.

[52] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "Todo or to bug," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 251–260.

[53] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments? */," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[54] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 11–20.

[55] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 260–269.

[56] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.

[57] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[58] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[59] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.

[60] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 215–223.

[61] J. Zhai, X. Xu, Y. Shi, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "Cpc: Automatically classifying and propagating natural language comments via program analysis," 2019.

[62] M. Y. S. L. Zhongxin Liu, Xin Xia, "Automating just-in-time comment updating," in *The 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[63] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.