Empirical Software Engineering (2024) 29:58
https://doi.org/10.1007/s10664-023-10425-5

Taxonomy of inline code comment smells

Elgun Jabrayilzade[1] · **Ayda Yurtoğlu**[1] · **Eray Tüzün**[1]

Accepted: 14 November 2023 / Published online: 3 April 2024

Abstract
Code comments play a vital role in source code comprehension and software maintainability. It is common for developers to write comments to explain a code snippet, and commenting code is generally considered a good practice in software engineering. However, low-quality comments can have a detrimental effect on software quality or be ineffective for code understanding. This study aims to create a taxonomy of inline code comment smells and determine how frequently each smell type occurs in software projects. We conducted a multivocal literature review to define the initial taxonomy of inline comment smells. Afterward, we manually labeled 2447 inline comments from eight open-source projects where half of them were Java, and another half were Python projects. We created a taxonomy of 11 inline code comment smell types and found out that the smells exist in both Java and Python projects with varying degrees. Moreover, we conducted an online survey with 41 software practitioners to learn their opinions on these smells and their impact on code comprehension and software maintainability. The survey respondents generally agreed with the taxonomy; however, they reported that some smell types might have a positive effect on code comprehension in certain scenarios. We also opened pull requests and issues fixing the comment smells in the sampled projects, where we got a 27% acceptance rate. We share our manually labeled dataset online and provide implications for software engineering practitioners, researchers, and educators.

Keywords Code comments · Comment smells · Inline code comments · Dataset · Taxonomy · Multivocal literature review

Communicated by: Simone Scalabrino, Rocco Oliveto, Felipe Ebert, Fernanda Madeiral and Fernando Castor

This article belongs to the Topical Collection: *Special Issue on Code Legibility, Readability, and Understandability*.

✉ Elgun Jabrayilzade
elgun1999@gmail.com

Ayda Yurtoğlu
ayda.yurtoglu@ug.bilkent.edu.tr

Eray Tüzün
eraytuzun@cs.bilkent.edu.tr

[1] Bilkent University, Ankara, Turkey

1 Introduction

Code comments are one of the primary sources helping software maintenance. They are the most used documentary artifact after the code itself (de Souza et al. 2005). Commenting has been generally recognized as a good practice in software engineering (Tenny 1988; Jiang and Hassan 2006), and developers usually write them to explain a particular code snippet. Code comments can generally be categorized into two: documentation and inline comments (Nielebock et al. 2019). Documentation comments are written above methods or classes to describe the functionality of the method or class. Documentation comments help developers easily understand how the method or class should be used, what are the parameter types, what is the return type, etc. Inline comments, on the other hand, are written inside a method or class body; thus, their scope is usually smaller than documentation comments. Inline comments can be written for a wide range of reasons, including code explanation, the reason behind a specific decision, etc.

Several studies in the literature analyzed the effect of code comments on program comprehension and software maintainability. Woodfield et al. (1981) conducted an experiment on 48 experienced developers by showing them different types of modularized codes with and without comments. The follow-up 20-question quiz revealed that the subjects whose code contained comments were able to answer more questions correctly. In agreement to this finding, 66% of software practitioners who responded to our survey in this study agreed that comments help them understand the code better. Similar to the findings of studies conducted by Ko et al. (2006) and LaToza et al. (2006), our survey results (see Section 4) reveal that developers spend as much time reading the code as writing it. Thus, having high-quality comments is crucial for software as they improve the legibility, readability, and understandability of code. Hartzman and Austin (1993) concluded that comments are a key factor in maintaining software in the long term in their case study on a large software system. Misra et al. (2020) found out that the lack of relevant comments in a project leads to a higher average issue resolution time in their empirical study on 625 GitHub repositories.

Since code comments contain abundant information about the software, researchers have also incorporated them into various tools for solving software engineering tasks, such as automated testing (Goffi et al. 2016; Wong et al. 2015), specification inference (Blasi et al. 2018; Pandita et al. 2012; Zhong et al. 2009), and code synthesis (Allamanis et al. 2015; Gvero and Kuncak 2015; Zhai et al. 2016). Having high-quality comments may improve the performance of such tools.

Because the compilers ignore comments and developers can write them in various ways, researchers have observed quality issues (smells) in comments (Louis et al. 2018; Khamis et al. 2010; Tan et al. 2007; Misra et al. 2020; Rani et al. 2021; Wang et al. 2019), such as being redundant or inconsistent with the code. Our study focuses on the analysis of inline code comment smells. In this context, smell refers to comments that can degrade software quality or comments that do not actually help readers much in terms of code comprehension. Although there have been studies on various categorizations of code comments and quality issues in them (discussed in Section 2), to the best of our knowledge, there is no previous work on the taxonomy and thorough analysis of inline code comment smells. This study extends our prior work on the analysis of inline code comment smells (Jabrayilzade et al. 2021), where we defined the following research questions:

- **RQ1.** ***What types of inline comment smells are there?***
- **RQ2.** ***How often does each smell type occur in practice?***

To answer the RQ1, we conducted a multivocal literature review (MLR) on academic (white) and gray literature and collected a set of inline comment smells written by developers. We proposed a taxonomy of 11 inline comment smells. For addressing the RQ2, we manually categorized a subset of inline comments from three open-source Java projects and found out that smells exist in the projects with varying degrees.

In this study, in order to make the results of RQ2 in the previous study [45] more generalizable, we added one more Java and four Python open-source projects to our analysis, increasing the total number of projects to eight. We found out that the smells exist in Python projects as well. Moreover, to gather practitioners' views on our defined taxonomy of inline comment smells as well as their impact on code comprehension and software maintainability, we added the following research question to this study:

– **RQ3.** ***What is the perception of software practitioners about inline code comment smells?***

To answer this research question, we conducted an online survey with 41 software practitioners. The respondents generally agreed with our taxonomy; however, there were smells for which respondents indicated that their existence might not be considered a smell in specific situations.

Finally to investigate the developer's perception of inline comment smells and whether or not they find them useful to code comprehension, we included a new research question in our study:

– **RQ4.** ***Do developers of open-source projects accept pull requests or issues to remove inline comments smells?***

We submitted 53 pull requests and/or issues to the projects we labeled to answer this question. We obtained an acceptance rate of 27%. Although removing inline comment smells is inherently beneficial to the code base, open-source project developers tend to reject pull requests that do not directly fix bugs and resist altering functional source code.

The extensions in this study over our previous work (Jabrayilzade et al. 2021) are as follows:

– Updated the methodology and scope of MLR results.
– Enhanced our evaluation by adding five new projects.
– Added a new research question (RQ3) to gather practitioners' views on our taxonomy of inline comment smells.
– Added a new research question (RQ4) to investigate the developers' acceptance of inline comment smells in eight OSS projects.

The main contributions of the study can be summarized as follows:

– Proposed a novel taxonomy of inline code comment smells based on a multivocal literature review and manual analysis of sample Java and Python projects.
– Manually analyzed four Java and four Python projects to determine how commonly the smells occur in open-source projects.
– Published the labeled dataset of inline comments consisting of 2447 comments based on the manual analysis process on the selected projects.
– Gathered practitioners' opinions about the inline comment smells through an online survey.
– Investigated developers' opinions about removing the inline comment smells from open-source projects with pull request submissions.

The rest of the paper is organized as follows. Section 2 presents the prior work on the analysis of code comments from different perspectives. Section 3 details our methodology for answering the research questions. Section 4 presents the results of the multivocal literature review, manual analysis process, and the practitioners' survey. Section 5 revisits the research questions, compares the results of the manual analysis process and survey responses, and provides implications for researchers, practitioners, and educators. Section 6 discusses the threats to the validity of the study. Finally, Section 7 concludes the work and presents the future directions of the study.

2 Related Work

We discuss the related works under three categories: general comment quality analysis, taxonomies of comments, and studies focusing on the detection of single code comment smell type.

2.1 Quality Analysis of Code Comments

In this section, we discuss works done previously about analyzing the quality of code comments from different perspectives.

Khamis et al. (2010) developed the tool JavadocMiner that can analyze the quality of Javadoc comments. The tool uses various heuristic metrics to assess the quality of the language used in comments and the consistency between source code and comments. The authors evaluated the tool on two Java projects (ArgoUML and Eclipse) by correlating the number of bugs in the modules of the projects and the metric scores. They report that modules with the highest quality had the lowest number of reported defects. Similarly, Tan et al. (2007) explored the feasibility and benefits of automatic analysis of comments to detect bugs and bad comments in source code. Their heuristic analysis on lock-related comments in the Linux kernel detected 12 bugs, two of which were confirmed by the kernel developers. The authors also examined various open-source bug datasets and found that bad or inconsistent comments introduced various types of bugs.

Rani et al. (2021) conducted a study on whether developers follow commenting guidelines in open-source projects. According to the results of the manual analysis done on 700 class comments from 13 Java and Python projects, they concluded that, in both languages, developers follow the writing style (e.g., grammar, punctuation) and content-related comment guidelines more often than syntax (denoting a specific type of a comment) and structure (e.g., subsections in comments) types of comments.

Misra et al. (2020) explored the correlation between code comments and issues in open-source Python projects from GitHub. They first extracted 20,000 comments from 625 repositories and then semi-automatically classified 600 of those comments into relevant and auxiliary comments. Relevant comments included header, method, inline, task, and section comments. On the other hand, auxiliary comments consisted of comments such as copyright, license, or information about authors. The authors also extracted the closed and open issues from the repositories as well as calculated the average time taken (in days) to resolve the issues. Statistical analysis showed that there exists a correlation between the number of relevant comments and the number of closed issues in a repository. Moreover, repositories with a higher number of relevant comments had a lower issue resolution time.

2.2 Taxonomies of Code Comments

This section describes studies on the categorization of code comments.

Steidl et al. (2013) used machine learning techniques on Java and C/C++ projects to automatically categorize comments into seven different categories, namely copyright, header, member, inline, section, code, and task comments. Their classification algorithms were nearly 95% precise. Then they assessed four quality criteria, namely, coherence, usefulness, completeness, and consistency, to determine the quality of the comments. They first proposed two metrics for the coherence of comments and then evaluated both metrics by a survey of experienced developers.

Haouari et al. (2011) aimed to categorize comments by their position and frequency in the code with an automated approach. Then, they used 49 programmer subjects to classify the comments via their content and relevance. The subjects classified comments by their object, type, style, and quality. The study did not emphasize code comment smells and only included a quality metric that can tell whether a comment can increase code comprehension or not.

Pascarella and Bacchelli (2017) created a taxonomy of Java comments consisting of 6 top and 16 inner categories. Then, they manually analyzed six Java projects to determine how often the categories occur in practice. They found that the *metadata* and *purpose* comments were the most prominent ones. They trained supervised machine learning models for the automatic classification of comment types and achieved an average of 0.95 true positive rate for top-level categories and 0.85 rate for inner categories. However, this study only categorized comments according to their purposes and did not include a qualitative analysis.

Zhang et al. (2018) investigated seven Python open-source projects and created a taxonomy for code comments consisting of 11 categories. The authors manually labeled 330 comments, which were randomly sampled from the selected projects to create a dataset of Python comments. Usage type was the most predominant category covering almost 80% of the labeled comments. They also applied machine learning classifiers to the dataset for classifying the code comments automatically. Their results showed that the Decision Tree classifier was superior to the Naïve Bayes classifier in terms of accuracy (87% compared to 81%) and runtime efficiency.

Shinyama et al. (2018) focused on identifying and classifying local comments that led to 11 distinct categories for code comments. With a manual annotation experiment, they verified the relevance of their comment categories and found out that *preconditional* and *postconditional* comments were dominant in their selected projects. They also used a decision-tree-based classifier to automatically analyze the code comments, which had 60% precision and 80% recall scores.

Khan et al. (2021) focused on the classification and automatic detection of API documentation smells. Firstly, they created 5 smell categories by consulting the literature, and then they created a benchmark with 1000 API documentation units where at least 778 documentation units belong to one of the smell categories. A survey with 21 professional software developers validated the catalog of API documentation smells by reporting that 95% of them think those smells impact their productivity negatively. Lastly, they used machine learning models to detect the five smells automatically, where the BERT (Devlin et al. 2019) classifier had the highest F1 scores.

Wen et al. (2019) conducted a study on code-comment inconsistencies to investigate how different code changes trigger comment updates and what types of code-comment inconsistencies are fixed by developers. Authors manually labeled 500 commits from Java projects and created a taxonomy composed of 6 root categories related to code-comment inconsistencies,

which are *Application Logic*, *Code Design/Quality*, *Maintenance*, *Formatting/Readability*, *Copyright/License*, and *Others*.

Padioleau et al. (2009) manually labeled 1050 comments from three operating system projects (Linux, FreeBSD, OpenSolaris), where the authors analyzed the comments in four dimensions: what is inside the comment, and for whom, where, and when they are written, and created a taxonomy for each dimension.

Zhai et al. (2020) studied automatic propagation and association of comments to code entities using program analysis. To effectively apply propagation, they created a taxonomy of comments under four entities, namely, class, method, statement, and variable. For each of these entities, they analyzed the comments from five perspectives: *what*, *why*, *how-it-is-done*, *property* (e.g., pre/post-conditions), and *how-to-use*. The authors also experimented with various machine learning algorithms for automatically classifying the comments into the defined categories, where random forest and decision tree algorithms had the best performance.

Table 1 provides categories defined for code comments for the studies discussed above. The studies conducted by Pascarella and Bacchelli (2017); Zhang et al. (2018) and Khan et al. (2021) are the most relevant ones to our study. The difference between our study and the former two studies is that we focus on classifying comment smells, i.e. bad practices in commenting, whereas they focus on the general classification of comments. The latter study focuses on smells on documentation smells in contrast to inline code comment smells that we investigate in this study.

2.3 Detection of Code Comment Smells

Although there is no previous study that considers inline code comment smells from a general perspective, there are various studies on analyzing and detecting smelly comments.

Louis et al. (2018) developed a deep learning model based on a long-short term memory network (Hochreiter and Schmidhuber 1997) to detect redundant (comments that restate the code) method-level comments. The model was trained in an unsupervised manner by measuring the predictability of the given comment sentence under the language model. In an

Table 1 Prior work on the taxonomy of code comments

Study	Categories
Steidl et al. (2013)	copyright, header, member, inline, section, code, task
Haouari et al. (2011)	object, type, style, quality
Pascarella and Bacchelli (2017)	summary, expand, rationale, deprecation, usage, exception, todo, incomplete, commented code, directive, formatter, licence, ownership, pointer, auto-generated, noise
Zhai et al. (2020)	metadata, summary, usage, parameters, expand, version, development notes, todo
Shinyama et al. (2018)	postcondition, precondition, value description, instruction, guide, interface, meta information, comment out, directive, visual core, uncategorized
Khan et al. (2021)	tangled, fragmented, lazy, bloated, excess structual information
Wen et al. (2019)	application logic, code design/quality, maintenance, formatting/readibility, copyright/licence, others
Padioleau et al. (2009)	what, who, where, when
Zhai et al. (2020)	what, why, how-it-is-done, property, how-to-use

experiment on Java comments, the authors found out that the Javadoc tags had a predictability score two times more than other comments.

Blasi et al. (2021) created a tool called RepliComment for identifying comment clones in code. The tool can report instances of copy-and-paste errors in comments as well as poorly written comments. The tool achieved 79% precision in finding critical clones according to the manual analysis done by the authors on 412 issues reported by the tool from various Java projects.

There are also studies on detecting which part of the code needs comment. Louis et al. (2020) developed a deep learning-based model that achieved 74% precision in finding comment-worthy locations in code. The authors created the evaluation dataset by extracting codes that were already commented. However, the approach assumed that the existing comments had high quality, which may bias the trained model in the case of smelly comments. A similar study was conducted by Huang et al. (2019), where they applied machine learning techniques to code context-related features. The effectiveness of the approach was evaluated via a survey of practitioners. The overlapping rate was 0.72 between the comment locations predicted by the model and the practitioners.

There have been multiple studies on identifying code-comment inconsistencies (*misleading* comments) in the literature. The models/tools either predict such comments statically by analyzing the relations between code and the corresponding comment (Sridhara 2016; Rabbi and Siddik 2020; Iammarino et al. 2020; Corazza et al. 2018; Wang et al. 2019; Ratol and Robillard 2017) or during a change (i.e., commit) (Panthaplackel et al. 2021; Liu et al. 2020, 2021; Stulova et al. 2020), which can also take change-related features into account.

3 Methodology

In this section, we describe the mixed-methods research design for answering the research questions. Specifically, we present the MLR stages and elaborate on the quantitative analysis performed on sample projects.

3.1 RQ1. What Types Of Inline Comment Smells Are There?

To answer RQ1, we first reviewed two books written by expert software practitioners, Clean Code by Martin (2009) and The Art of Readable Code (Boswell and Foucher 2011) by Dustin Boswell and Trevor Foucher to get an initial idea about bad practices followed by developers when writing code comments. We encountered various types of comments that affect the readability and maintainability of source code. Due to the lack of research in academia and the abundance of gray literature materials (e.g., blog posts) published by software practitioners on this topic, we decided to carry an MLR according to the guidelines provided by Garousi et al. (2019). The activities conducted during the MLR are given in Fig. 1. The execution of search queries, applying inclusion/exclusion criteria, and extracting relevant smell data were conducted by two authors independently. Conflicting cases were resolved in an additional online meeting between the two authors. The MLR was conducted between 4 - 18 March 2021.

Search Strategy Google Scholar and Scopus were used for searching the academic literature as they include major publication databases (Neuhaus et al. 2006). For gray literature, the authors utilized the Google Search engine according to the suggestions of Godin et al. (2015).

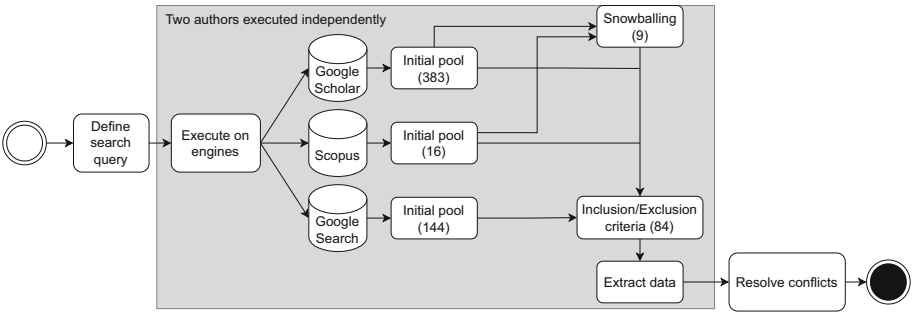


Fig. 1 Flow diagram of the MLR stages

We defined the following search query string to retrieve materials relevant to the research question.

> ***"Code Comment"*** *AND* (***"Guideline"*** *OR* ***"Smell"*** *OR* ***"Bad Practice"*** *OR* ***"Anti-pattern"***)

The search query was defined according to previous MLR studies on identifying various anti-patterns in software engineering and we selected keywords that can capture relevant sources (Garousi and Küçük 2018; Qamar et al. 2022; Doğan and Tüzün 2022). All the results provided by the search engines were analyzed thoroughly. The authors applied inclusion/exclusion criteria to only include sources related to the code comment smells. After creating the initial pool of sources, we also manually conducted backward and forward snowballing (checking references and citations of papers) on the academic papers not to miss any related studies, as suggested by Keele et al. (2007). Backward snowballing is the application of inclusion/exclusion criteria to the references of a paper that is already in the pool of sources. In forward snowballing, we applied the inclusion/exclusion criteria for the studies citing a paper that is already in the pool of sources. We iteratively applied backward and forward snowballing until there were no more primary studies to be added.

Inclusion/Exclusion Criteria We carefully designed the inclusion/exclusion criteria to incorporate relevant and high-quality sources. We included the sources (both white and gray) that answered "yes" to all the following questions:

- Does the source discuss bad or good practices about code comments?
- Is the source in English, and is it fully accessible?
- Does the source contain author information?
- Does the source contain non-duplicate information? (not restating another source)
- Is the source published in a peer-reviewed conference or journal? (only white literature)

Final Pool of Sources Applying the search query returned 383 sources from Google Scholar, 16 from Scopus, and 144 from Google Search. To avoid missing any studies, the two authors performed separate searches on the selected databases and applied inclusion/exclusion criteria to the search results. When there was a disagreement on the selection of the primary studies, the third author was consulted, and the problems were discussed until we came to a consensus. Additionally, we found nine sources from white literature as a result

of backward and forward snowballing. The final list consisted of 84 relevant resources. Out of those sources, 55 of them were retrieved from gray literature and 29 from white literature.

Data Extraction After we compiled a list of 84 studies identified in the previous phase, two authors independently analyzed these studies. We used a shared spreadsheet to encode the specific smell type we encountered in the studies by giving it a name. Upon adding the smell type to the spreadsheet, each author could either use one of the previously defined categories or add a new one. After we achieved the initial set of categories, we merged the related categories. The categories were finalized after a few rounds of discussions among the authors. The table of the sources and discussed smell types in those sources is available online.[1]

3.2 RQ2. How Often Does Each Smell Type Occur In Practice?

To answer how frequently code comment smells appear in software engineering practice, we manually analyzed inline comments of four Java and four Python projects from GitHub. The following sections describe the project selection, comment sampling, and manual labeling processes.

Project Selection Because there was a large number of projects to choose from, we filtered them first. We selected the top 1000 Java and top 1000 Python projects according to the number of GitHub stars to include popular projects used by the community. Next, in order to remove personal or small projects, we filtered out projects that had less than ten contributors. We then randomly selected projects having heterogeneous domains (e.g., mobile applications) and different project sizes (in terms of lines of code). While choosing the projects, we also checked if the projects were in English (according to README files) and if they were real-life projects (e.g., non-education related). Moreover, we only included active projects - projects that had at least one commit in the last month. In the end, we selected *Anki-Android* (a flashcard memorization app),[2] *Jitsi* (video conferencing service),[3] *Moshi* (JSON library),[4] and *Light-4j* (microservices framework)[5] projects for Java, and *Requests* (HTTP library),[6] *Scrapy* (web crawling framework),[7] *Kivy* (UI framework),[8] and *Scikit-learn* (machine learning framework)[9] projects for Python. The selected projects with their GitHub stars, the number of contributors, and the lines of code (LOC) are shown in Table 2. LOC shows only code lines (non-empty, non-comment) written in the project's main language (either Java or Python).

Comment Sampling We used a comment parser tool[10] to find the comment locations in the Java source files of the projects. We filtered out documentation comments (beginning with "/**") to include only inline comments. For Python, we located the comments by searching

[1] https://doi.org/10.6084/m9.figshare.19640886.v9

[2] https://github.com/ankidroid/Anki-Android

[3] https://github.com/jitsi/jitsi

[4] https://github.com/square/moshi

[5] https://github.com/networknt/light-4j

[6] https://github.com/psf/requests

[7] https://github.com/scrapy/scrapy

[8] https://github.com/kivy/kivy

[9] https://github.com/scikit-learn/scikit-learn

[10] https://github.com/jeanralphaviles/comment_parser

Table 2 Details of the selected projects

		Stars	Contributors	LOC	Inline comments	Sampled files	Sampled comments	Retrieval date
Java	Anki-Android	3.9k	248	129k	4749	12	355	2021-06-01
	Jitsi	3.4k	59	582k	9404	37	369	2021-06-17
	Moshi	7.4k	66	23k	321	21	175	2021-06-25
	Light-4j	3.3k	35	78k	1023	66	279	2021-10-11
Python	Requests	47k	606	8k	608	17	247	2021-11-19
	Scrapy	42.9k	446	41k	1294	40	296	2021-12-05
	Kivy	14.4k	444	86k	4082	28	351	2022-01-11
	Scikit-learn	49.2k	2247	266k	14965	22	375	2022-02-09

for hash symbols in the source code. The number of inline comments extracted from each project is presented in Table 2. Due to the large number of inline comments in the selected projects, we found it infeasible to analyze all of them manually. Instead, we randomly sampled a statistically significant subset of the comments from the selected eight projects and analyzed them. The sample size was determined via Cochran's sample size formula (Cochran 2007):

$$n = \frac{z^2 * p * (1-p)/e^2}{1 + (\frac{z^2 p*(1-p)}{e^2 N})}$$

The sampling was done with a 95% confidence level and 5% margin of error. We chose $p = 0.5$ as it is unknown which proportion of comments are smell or not. The number of sampled files and comments for each project is shown in Table 2. For each project, we randomly sampled code files (ending with ".java" or ".py") and selected all the inline comments from that file. The files were sampled until the required sample size was satisfied. The reason for using file granularity was to make it easier for the authors to label the comments with high confidence as the context of comments can be difficult to understand.

Comment Labeling Process We developed a web-based tool to label the sampled comments manually. A sample snapshot of the tool is shown in Fig. 2. The tool is user authenticated so that the labelers cannot see each others' labels. The tool presents one comment at a time and requires the labeler to categorize it into one of the smell types. Labelers are able to scroll through the code snippet, see the number of labeling they have done, edit previous labels and see the available smell categories. The tool also has a section for labelers to add their opinions about the given comment if there is something that needs to be taken into consideration. Additionally, users are required to provide information about their confidence in categorization as well as the related code part that the comment was pointing to.

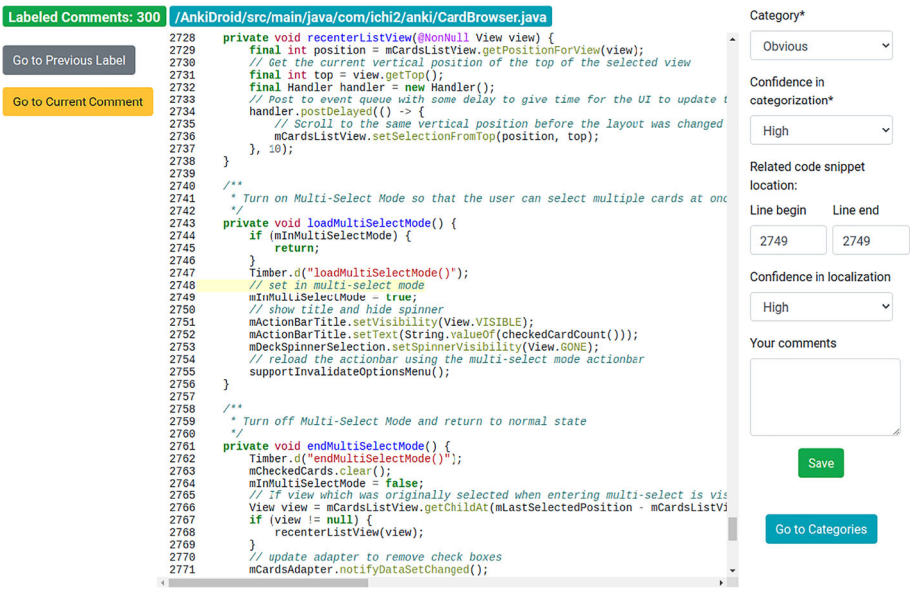


Fig. 2 Snapshot of the tool developed for labeling the comments

The labeling process was executed by two authors independently. The authors then conducted a meeting to resolve the conflicts. In the case of disagreement, the third author was asked to provide categorization, and the final labels were determined by voting.

Before labeling the selected projects, we sampled a hundred comments from Hive[11] project to test the application and determine the average labeling time for a single comment (which was about a minute). During this process, we found an additional smell type (*vague* comments) that was not encountered during the MLR. The details of this smell type are discussed in Section 4.

3.3 RQ3. What is the Perception of Software Practitioners About Inline Comment Smells?

We have conducted an online survey through Google Forms to answer this research question. The survey's goal was to determine whether software practitioners agree with our taxonomy of inline comment smells and to gather their opinions about the effect of the smells on code maintainability or comprehension.

Survey Protocol Our survey consisted of two parts. In the first part, we asked demographic questions as well as the proportion of their daily work time spent on code writing, reading, and reviewing activities. Specifically, we asked the following questions:

1. What is your current job title? [Text]
2. For how many years have you been working in the software industry? [Likert scale]
3. What percentage of your daily work time do you spend on writing code? [Likert scale]
4. What percentage of your daily work time do you spend on reading code? [Likert scale]
5. What percentage of your daily work time do you spend on reviewing code? [Likert scale]

In the second part, we first asked whether they agreed that inline code comments help them understand the code better (Likert scale). Afterward, for each smell type, we showed its description with a sample example and asked whether they agreed that the given type of inline comment is a smell or not, alongside their reasoning. We also asked how often they see such kind of comment and their opinion about its impact on software maintainability and code comprehension. Questions asked in the second part of the survey for each smell type are listed below:

1. Do you agree that the given type of comment is a smell and should not be written in the source code? [Likert scale]
2. Can you please explain your reasoning? [Text]
3. How often do you encounter this type of comment? [Likert scale]
4. In your opinion, how does this comment type impact the software maintainability/code comprehension? [Likert scale]

At the end of the survey, we had an open-ended section asking whether they had anything else to say about the smells or a new smell category they would like to add.

Recruitment We recruited participants through the networks of the authors and posted the survey on Twitter and LinkedIn platforms. Respondents participated in a raffle consisting of a $100 Amazon Gift Card. Before publishing the actual survey, we conducted a pilot test with five engineers from the authors' network to ensure that the survey was clear and understandable. During the pilot test, we got feedback on the completion time (≈12 minutes

[11] https://github.com/apache/hive

on average) of respondents and some survey parts being ambiguous. We addressed ambiguous parts in the actual survey. We did not include the responses to the pilot test in the survey results.

3.4 RQ4. Do Developers Of Open-Source Projects Accept Pull Requests Or Issues To Remove Inline Comments Smells?

We manually submitted a total number of 53 pull requests and issues to the eight projects that we used during labeling. Before submitting any pull requests, we first checked the contribution guidelines of each chosen project to make sure we did not violate the pull request structure.

For each chosen project, we opened pull requests and issues to fix the comment smells. The smells were chosen from the comments that we manually labeled in a way that we had at least one smell from each category (except *attribution* smell type since we did not encounter it in our labeled projects). The comments in the categories, on the other hand, were sampled randomly. By randomly choosing comments from different projects, we aimed to ensure that the comment smells represent a diverse set of scenarios found in real-world software projects. This approach helped us to capture various types of inline comment smells, including both straightforward and complex cases, which may require different degrees of understanding and effort to fix.

In each pull request, we suggested removing or shortening a comment smell. We explained the smell type and our reasoning for such a change. The pull requests we submitted consisted of different types of comment smells. However, we had different approaches while removing each type of comment. For *obvious*, *beautification*, *commented-out code*, and *irrelevant* comments, we proposed to remove the whole comment whereas comments with too much information were shortened considerably. *Misleading* comments were handled by creating a pull request that fixed the comment to be aligned with the code. For *task* smells, we opened a pull request to remove the comment from the code and created an issue in GitHub instead regarding that task. For *vague* comments, we opened pull requests if we understood the issue and the fix was trivial. Otherwise, we opened issues by asking the developers to elaborate and explain the comment. For *non-local information* smells, we opened pull requests where we moved the comment to its correct place in the code. The pull requests and issues that we opened are available in our replication package.

It is unconventional to modify different parts of the codebase at the same time, hence; we submitted our pull requests for each smell separately. Generally, modifying the unrelated parts of the codebase in a single pull request is also not welcomed by the developers. Additionally, the developers of open-source projects are likely to reject pull requests that do not fix a certain issue, so we avoided submitting too many pull requests to one project [4]. After pull request submissions, we counted the number of accepted and rejected pull requests and the ones that did not receive any feedback to report the results.

4 Results

This section discusses our findings as a result of the multivocal literature review, the manual labeling process of eight projects, and the practitioner's survey.

Table 3 Taxonomy of inline code comment smells

Smell type	Description	Example	Sources

Misleading	Comments that do not accurately represent what the code does	public int add(int x, int y){
		// Returns x - y
		return x + y;
		}
			45

Obvious	Comments that restate what the code does in an obvious manner	count = 0; // assigning count a value of 0	41
No comment on non-obvious code	Non-obvious piece of code is left without a comment	if (count > 55 && count < 70) { ... }	26
Commented-out code	A code piece that is commented-out	//facade.registerProxy(newSoundAssetProxy());	18
Irrelevant	Comments that do not intend to explain the code	/* I dedicate all this code, all my work, to my wife, Darlene, who will have to support me and our children and the dog once it gets released into the public.*/	10
Task	Comments explaining the work that could/should be done in future or was already completed	// TODO: clear and optimize this code later	9
Too much information	Overly verbose comments	// this makes a new scanner, which can read from // STDIN, located at System.in. The scanner lets us look // for tokens, aka stuff the user has entered. Scanner sc = new Scanner(System.in);	8
Attribution	Comments that give information about who wrote the code	/* Added by Rick */	7
Beautification	Comments that aim to distinguish the parts of the code	//**********// VARIABLES //**********	6
Non-local	Comments that provide systemwide information or mention code that is not near	public void setFitnessePort(int fitnessePort) { // Port on which fitnesse would run. Defaults to 8082 this.fitnessePort = fitnessePort; }	3
Vague	Comments that are not clearly understandable	taskTmpPath = ttp; //_task_tmp	NA

4.1 Final Taxonomy (RQ1)

The multivocal literature review led to ten types of source code commenting practices that are considered as a smell by the practitioners. We also found an additional smell type called *vague* comments during the labeling process of Java projects and observed this type of smell occurring in Python projects as well. We did not receive any other smell type suggestions from the practitioner's survey. Table 3 shows the final taxonomy of these 11 smell types alongside their description, an example, and the number of sources that they were mentioned. *Misleading* and *non-local* smell types were mentioned the most and the least in our MLR sources, respectively. We present each smell type in the subsections below. We also provide example smells from the projects we labeled.

Misleading Comments that do not accurately represent what the code does are considered misleading comments. This type of smell usually happens after a developer makes a code change but forgets to update its related comment, which creates code-comment inconsistencies and could misguide the readers. This can also occur when the author forgets to update comments of a copy-pasted code. Figure 3 shows an example of a *misleading* comment (mentioning the removal of a file named *test.json*, however, the file is named differently in the following code piece).

Obvious Comments that restate what the code does without giving additional information are considered obvious comments. Such comments decrease the readability and clutter the source code. An example of *obvious* smell is given in Fig. 4, where the comment restates the error message in the line below.

No Comment on Non-Obvious Code According to the practitioners, it is generally a good practice to write comments to state the reasoning behind the code instead of what it does. Lack of comments on complex parts of the source code decreases understandability and affects software maintenance. We do not present an example for this smell type since we ignored such smells during the labeling process.

Commented-out code A code piece that is commented out is referred to as a commented-out code. Developers usually comment out code snippets for debugging purposes. However, such comments should not be pushed to the codebase as they clutter the source code and are generally ignored by the readers. An example of this smell from the *Light-4j* project is given in Fig. 5.

Irrelevant Irrelevant comments are unnecessary comments that do not intend to and do not, provide any information on the code or project. Figure 6 shows a sample *irrelevant* comment from the *Requests* project.

Task Task comments explain the potential future work or the work that was already completed. These are often "TODO" or "FIXME" types of comments. Task comments can also include comments written without such keywords (e.g., "*should we optimize this?*").

```
// Remove the test.json from home directory
File configFile = new File(homeDir + "/info.yml");
configFile.delete();
```

Fig. 3 Example of *misleading* smell from the Light-4j project

```
# make sure flags list is shallow copied
assert r1.flags is not r2.flags, "flags must be a shallow copy, not identical"
self.assertEqual(r1.flags, r2.flags)
```

Fig. 4 Example of *obvious* smell from the Scrapy project

Although most modern code editors highlight task comments with keywords, developers sometimes forget to remove them even if the task is completed Storey et al. (2008). They often pile up and litter the source code. It is generally suggested to define such tasks in a project/issue management tool. Figure 7 shows an example *task* comment about optimizing a piece of code.

Too much Information This type of comment smell happens when a comment is overly verbose. Such comments often contain historical information or are very detailed but do not provide meaningful insights, which decreases readability. An example of this smell type is given in Fig. 8.

Attribution Comments that give information about who wrote the code are referred to as attribution comments. Such comments are unnecessary as this information can be retrieved from most version control systems such as Git.[12] We did not find an occurrence of *attribution* smell in our labeled projects.

Beautification Sometimes, developers want to distinguish parts of a source code. They usually use a sequence of arbitrary characters to indicate those parts. These are also often called separators. Such comments may create distractions and are generally considered unnecessary. An example of *beautification* smell from the *Anki-Android* project is presented in Fig. 9, where the comment is used for indicating the location of class methods.

Non-Local Comment should describe the code near to it so that the readers can easily understand the source code. We call comments that give systemwide details or mention code part that is far away from the comment location as non-local comments. Figure 10 shows an example of this smell, where the comment mentions a call to a procedure during a specific event, which is not related to the code nearby.

Vague One of the reasons behind writing code comments is to make readers easily understand the code. Thus, comments should also be intelligible. We label comments that are difficult to comprehend as vague comments. An example of *vague* comment from the *Kivy* project is presented in Fig. 11.

Summary of RQ1: What types of inline comment smells are there?

We identified 11 categories of inline comment smells, which are *misleading*, *obvious*, *no comment on non-obvious code*, *commented-out code*, *irrelevant*, *too much information*, *task*, *attribution*, *non-local*, *vague* and *beautification* smells through a multivocal literature review and a practitioner survey.

[12] https://git-scm.com/

```
latch.await();

// final ClientResponse response = reference.get();
Assert.assertNotNull(reference.get().getAttachment(Http2Client.BUFFER_BODY));
Assert.assertEquals(false, connection.isOpen());
```

Fig. 5 Example of *commented-out code* smell from the Light-4j project

4.2 Empirical Results (RQ2)

We discuss the results of the manual labeling process on eight Java and Python projects in this section. In the first stage of the process, two authors independently labeled the sampled comments followed by a conflict resolution meeting. The initial agreement was 0.53 between the two authors according to Cohen's kappa score (Cohen 1960) and increased to 0.94 after the meeting. The remaining conflicts were resolved in an additional meeting with the third author. Table 4 shows the number of comments we found for each smell type. We excluded the smell type *no comment on non-obvious code* during our labeling process as it requires analyzing the code files line by line, and determining whether a given code piece needs comment is not a trivial task. The distribution of comments and smells are shown in Figs. 12 and 13, respectively. The labeled inline comment smell dataset is available online.[13]

The manual analysis of eight projects led to a considerable number of comment smells in the selected projects, and the most observed smell type was *obvious* comments. This smell type occurred the most out of all smells in all eight projects. *Task* comments had the second highest number of occurrences in seven out of eight projects. It only ranked third in Jitsi, a Java project, where it was surpassed by *commented-out code* comments.

We have observed all types of smells, except for *attribution* which had no occurrences at all, in at least two projects. After this smell type, the *irrelevant* and *non-local* comments had the least occurrences by only occurring in two out of eight projects. Comments with *too much information* occurred only in four projects. Hence, these four types of comment smells were observed considerably less than other types. The other smell types were observed occasionally, although their ratio differs from project to project.

The projects with a higher contributor count (>400) have a higher ratio of proper comments (comments that are not smells) than other projects. These projects also have higher star counts (>40.000) than the others. We also observed that all three projects (Requests, Scrapy and Scikit-learn) were developed in Python. Hence, the Python projects that we analyzed had lower percentages of comment smells than Java projects.

There are some smell types whose ratio differs significantly in projects (e.g., *task*). To investigate this, we analyzed GitHub repositories of the projects to see if there were guidelines about code commenting. We found instructions about writing *task* and *beautification* comments in *Anki-Android* guidelines,[14] as shown in Fig. 14. Moreover, *Jitsi* guidelines[15] points to the old Java code conventions that reference *commented-out code*. On the other hand, Scikit-learn has a different approach in its code guidelines[16] where it encourages avoiding *obvious*, *vague*, and *irrelevant* comments and adding comments where necessary.

[13] https://doi.org/10.6084/m9.figshare.19640886.v9

[14] https://github.com/ankidroid/Anki-Android/wiki/Code-style

[15] https://desktop.jitsi.org/Documentation/CodeConvention

[16] https://scikit-learn.org/stable/developers/contributing.html

```
target = chardet.__name__
for mod in list(sys.modules):
    if mod == target or mod.startswith(target + '.'):
        sys.modules['requests.packages.' + target.replace(target, 'chardet')] = sys.modules[mod]
# Kinda cool, though, right?
```

Fig. 6 Example of *irrelevant* smell from the Requests project

According to its code review guidelines, the questions that should be asked during code review are as follows: "*Is the code easy to read and low on redundancy? Should variable names be improved for clarity or consistency? Should comments be added? Should comments be removed as unhelpful or extraneous?*". Also, all the Python projects refer to the PEP 8 code style guide[17] in their contribution guidelines. Figure 15 shows a snapshot from the PEP 8 guidelines mentioning comments should not contradict the code (*misleading* smell), should be easily understandable (*vague* smell), and should not state the obvious (*obvious* smell). Although there is a resource on Java code conventions published by Oracle,[18] it was not mentioned in the contribution guidelines of the selected Java projects.

Summary of RQ2: How often does each smell type occur in practice?

We manually labeled sample comments from four Java and four Python projects and found out that nearly 44% of comments are smells with the *obvious* type of comment smell having the most occurrences. On the contrary, we did not find any occurrence of *attribution* smell.

4.3 Survey Results (RQ3)

Initially, we got 66 responses to our survey. Since this is an online survey with anonymous respondents, we filtered out potentially unrelated / low-quality responses and kept the responses whose job title was related to software engineering. For those that have non-software engineering titles, we also analyzed their responses and observed that they have consistently not answered the open-ended questions. Figure 16 shows the roles of the respondents. The majority of them were software engineers (we combined engineer and developer roles). We also got responses from practitioners with positions such as head of product, intern, postdoc, etc. The distribution of software engineering experience of the respondents can be seen in Fig. 17, where the majority had middle to senior-level engineering experience. Respondents' activities related to writing, reading, or reviewing code are shown in Fig. 18. They mostly spend their daily work on writing and reading code rather than conducting code reviews.

We asked a general question in the survey to practitioners on whether inline code comments help them understand the code better. The distribution of the responses is given in Fig. 19, where 66% of the respondents agreed that they are helpful for code comprehension.

We further evaluate the perceptions of the practitioners in three categories: smell taxonomy, smell occurrences, and impacts of smells.

[17] https://peps.python.org/pep-0008/

[18] https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

```
if (nearest != null) {
  // Micro-optimization: avoid polymorphic calls to Comparator.compare().
  @SuppressWarnings("unchecked") // Throws a ClassCastException below if there's trouble.
  Comparable<Object> comparableKey =
      (comparator == NATURAL_ORDER) ? (Comparable<Object>) key : null;

  while (true) {
    comparison =
        (comparableKey != null)
            ? comparableKey.compareTo(nearest.key)
            : comparator.compare(key, nearest.key);
```

Fig. 7 Example of *task* smell from the Moshi project

4.3.1 Perceptions on Smell Taxonomy

In our survey, we asked practitioners "Do you agree that the given type of comment is a smell and should not be written in the source code?" and their answer's reasoning for every smell type. They could choose an answer from the following options: *strongly disagree*, *disagree*, *neutral*, *agree* and *strongly agree* (Fig. 20).

We saw that practitioners mostly disagreed with *task* comments (51.2%) being considered as smells. Some of the reasoning behind *task* comments not being smells was that these type of comments can be useful during the development process as it reminds the engineer what task to do but should not be there in the final product. Others who agreed with it mentioned that *task* comments should be in the issue tracking software instead of the code. A couple of respondents made the following remarks about *task* comments:

```
def test_url_encoding_nonutf8_untouched(self):
    # percent-escaping sequences that do not match valid UTF-8 sequences
    # should be kept untouched (just upper-cased perhaps)
    #
    # See https://tools.ietf.org/html/rfc3987#section-3.2
    #
    # "Conversions from URIs to IRIs MUST NOT use any character encoding
    # other than UTF-8 in steps 3 and 4, even if it might be possible to
    # guess from the context that another character encoding than UTF-8 was
    # used in the URI.  For example, the URI
    # "http://www.example.org/r%E9sum%E9.html" might with some guessing be
    # interpreted to contain two e-acute characters encoded as iso-8859-1.
    # It must not be converted to an IRI containing these e-acute
    # characters.  Otherwise, in the future the IRI will be mapped to
    # "http://www.example.org/r%C3%A9sum%C3%A9.html", which is a different
    # URI from "http://www.example.org/r%E9sum%E9.html".
    r1 = self.request_class(url="http://www.scrapy.org/price/%a3")
    self.assertEqual(r1.url, "http://www.scrapy.org/price/%a3")
```

Fig. 8 Example of *too much information* smell from the Scrapy project

```
// ------------------------------------------------------------------------
// Class methods
// ------------------------------------------------------------------------

public static Intent getPreferenceSubscreenIntent(Context context, String subscre
    Intent i = new Intent(context, Preferences.class);
    i.putExtra(android.preference.PreferenceActivity.EXTRA_SHOW_FRAGMENT, "com.ic
    Bundle extras = new Bundle();
    extras.putString("subscreen", subscreen);
    i.putExtra(android.preference.PreferenceActivity.EXTRA_SHOW_FRAGMENT_ARGUMENT
    i.putExtra(android.preference.PreferenceActivity.EXTRA_NO_HEADERS, true);
    return i;
}
private void initSubscreen(String action, PreferenceContext listener) {
```

Fig. 9 Example of *beautification* smell from the Anki-Android project

> *"In big codebases, these todos never get done. It is better to create a ticket etc. But this can help the reader see incapabilities of the code which I think helps comprehensibility."*

> *"It's usually used to grab the attention of the developer as a reminder to complete this part of the project code."*

> *"TODO comments are helpful if they're used properly and maintained (handled). They can help other developers understand how the code will be evolving, or they can be helpful as reminders."*

Beautification types of comments were also disagreed by a number of respondents for being considered as a smell. Some reasons include that they help with organizing the code by making it more comprehensive visually. Most practitioners agreed that it makes the code more readable in cases where the code is too long or complex. However, one participant mentioned that these comments were necessary when IDEs were not sufficient at syntax highlighting, but since IDEs improved in the previous years, *beautification* comments are not needed anymore. The following comments are made by the respondents about the *beautification* smell:

```
//first store the account and only then load it as the load generates
//an osgi event, the osgi event triggers (through the UI) a call to the
//ProtocolProviderService.register() method and it needs to acces
//the configuration service and check for a stored password.
this.storeAccount(accountID, false);

accountID = loadAccount(accountProperties);
```

Fig. 10 Example of *non-local* smell from the Jitsi project

```
if words is not None and len(words) > 1:
    space = type(line)(' ')
    # words: every even index is spaces, just add ltr n spaces
    for i in range(n):
        idx = (2 * i + 1) % (len(words) - 1)
        words[idx] = words[idx] + space
```

Fig. 11 Example of *vague* smell from the Kivy project

> *"This is a nice thing to do when your code is seen by starters who just started coding, but in industry, there should not be a need to see this comment to understand variables are declared there, etc."*
>
> *"If there are styling rules (i.e., variables first, then constructors, functions, helpers, etc.), then it is not needed. But it may be helpful if the file is too long."*
>
> *"When the class or respective file is too large, which could also be another code smell (God class), then it might be okay to use such comments."*

Respondents who disagreed with the *non-local information* type of comment being smell mentioned that they may help unfamiliar developers understand how code works. On the other hand, we got responses pointing out that this type of comment should be written in the documentation, not in the source code, and may contain sensitive information.

For other types of smells, such as *commented-out code* and *too much information*, some practitioners argued that these comments might be helpful for developers. For *commented-out code* comments, some argued that they are useful during debugging but should not be there in the final product whereas some said they are not necessary because they can be retrieved from version control systems. For comments with *too much information*, some arguments said that they can be beneficial for people by making the explanation of code very clear. One response pointed out that such comments can be useful in open-source applications where there is no space to store documentation and historical information. However, many practitioners agreed that these comments are hard to maintain and should be put in the documentation.

Another smell type that received interesting responses was *irrelevant*. While 73.1% of the practitioners agreed that this is a smell, there were responses indicating that they can be entertaining sometimes. We got such remarks about *irrelevant* smells:

> *"Does not contain any information regarding development and is distracting."*
>
> *"I think it is a code smell but I personally would like to see these kinds of uplifting comments if they occur only in a few places."*

The respondents generally agreed with the other types of smells according to the number of *agree* and *strongly agree* responses. The statistics of the respondents' agreement on the comment smells are shown in Table 5.

4.3.2 Perceptions on Smell Occurrences

For every smell type in our taxonomy, practitioners were asked "How often do you encounter this type of comment?" in the survey. They were expected to choose an answer from the given

Table 4 Occurrences of the smell types in the selected projects

	Java				Python				Total
	Anki-Android	Jitsi	Moshi	Light-4j	Requests	Scrapy	Kivy	Scikit-learn	
Not a smell	182 (51.3%)	136 (36.9%)	82 (46.9%)	155 (55.6%)	154 (62.3%)	213 (72.0%)	214 (61.0%)	239 (63.7%)	1375 (56.2%)
Misleading	3 (0.9%)	8 (2.2%)	0 (0.0%)	3 (1.1%)	2 (0.8%)	1 (0.3%)	0 (0.0%)	2 (0.5%)	19 (0.8%)
Obvious	97 (27.3%)	193 (52.3%)	79 (45.1%)	88 (31.5%)	66 (26.7%)	49 (16.5%)	102 (29.1%)	84 (22.4%)	758 (30.1%)
Commented out code	8 (2.3%)	17 (4.6%)	0 (0.0%)	10 (3.6%)	1 (0.4%)	0 (0.0%)	0 (0.0%)	2 (0.5%)	38 (1.6%)
Task	36 (10.1%)	9 (2.4%)	6 (3.4%)	13 (4.7%)	13 (5.3%)	10 (3.4%)	15 (4.3%)	28 (7.5%)	130 (5.3%)
Too much information	1 (0.3%)	0 (0.0%)	0 (0.0%)	1 (0.4%)	0 (0.0%)	1 (0.3%)	2 (0.6%)	1 (0.3%)	6 (0.3%)
Attribution	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Irrelevant	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.4%)	3 (1.2%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	4 (0.2%)
Beautification	14 (3.9%)	0 (0.0%)	0 (0.0%)	4 (1.4%)	3 (1.2%)	9 (3.0%)	10 (2.9%)	10 (2.7%)	50 (2.0%)
Non-local	0 (0.0%)	2 (0.5%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (0.5%)	4 (0.2%)
Vague	14 (3.9%)	4 (1.1%)	8 (4.6%)	4 (1.4%)	5 (2.0%)	13 (4.4%)	8 (2.3%)	7 (1.9%)	63 (2.6%)
Total	355	369	175	279	247	296	351	375	2447

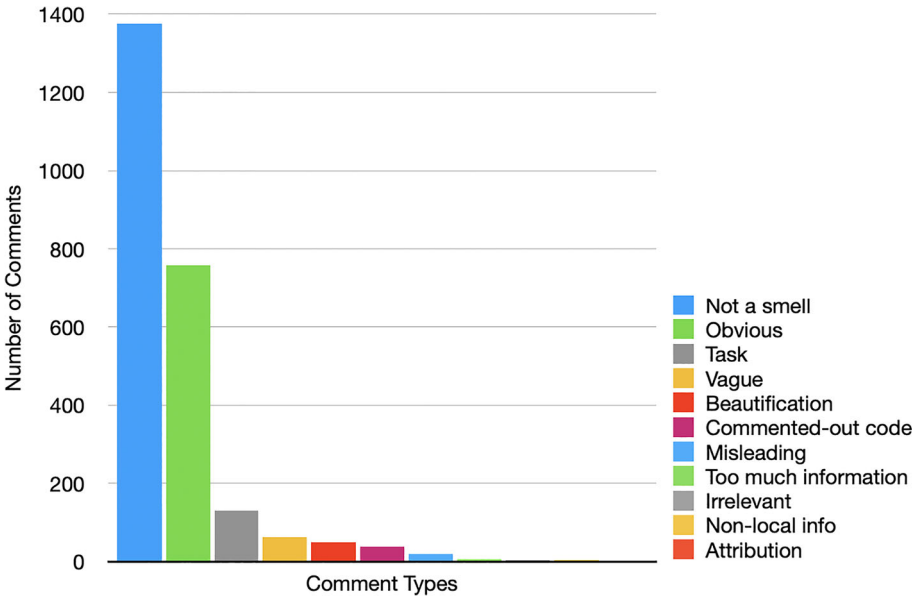


Fig. 12 Distribution of comment types (all projects combined)

options: *never*, *rarely*, *sometimes*, *often*, and *nearly always*. The distribution of responses can be seen in Fig. 21. According to their answers, we took the weighted average by giving the following weights respectively for each option: 0.0, 0.25, 0.5, 0.75, and 1. Then we ranked the smell types (see Table 6) according to their weighted average starting with the most encountered one. We omitted *no comment on non-obvious code* smell type from the table as we do not have data from our labeling process to compare to.

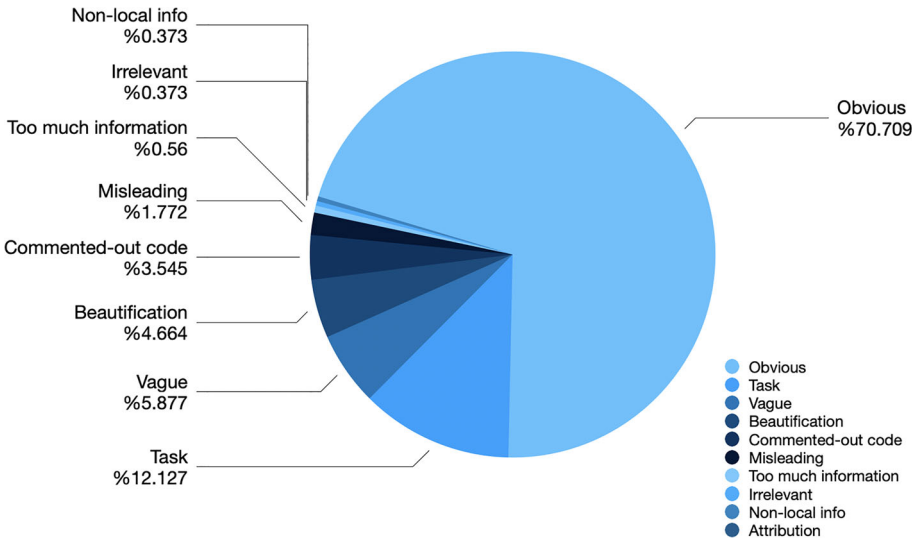


Fig. 13 Distribution of smells (all projects combined)

For separation comments within differents parts of a file, the following forms should be used (depending on its level of importance):

```
//*********************
//
//*********************
```

```
//---------------------
//
//---------------------
```

TODO and FIXME must be written all in capitals and followed by a colon.

- `// TODO: Calculate the new order` // NOT: ~~// TODO -> Calculate the new order~~
- `// FIXME: Fix the synchronization algorithm` // NOT: ~~fixme: Fix the synchronization algorithm~~

The TODO comment should be used to indicate pending tasks, code that is temporary, a short-term solution or good enough but not perfect code.

The FIXME comment should be used to flag something that is bogus and broken.

Fig. 14 A segment of Anki-Android code guidelines

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.

Ensure that your comments are clear and easily understandable to other speakers of the language you are writing in.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Fig. 15 PEP 8 guidelines on code comments

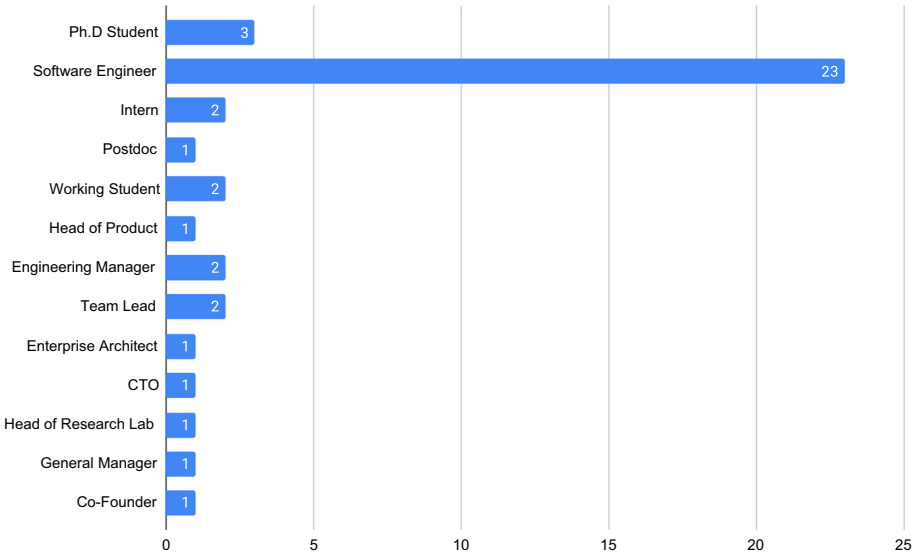


Fig. 16 Roles of survey respondents

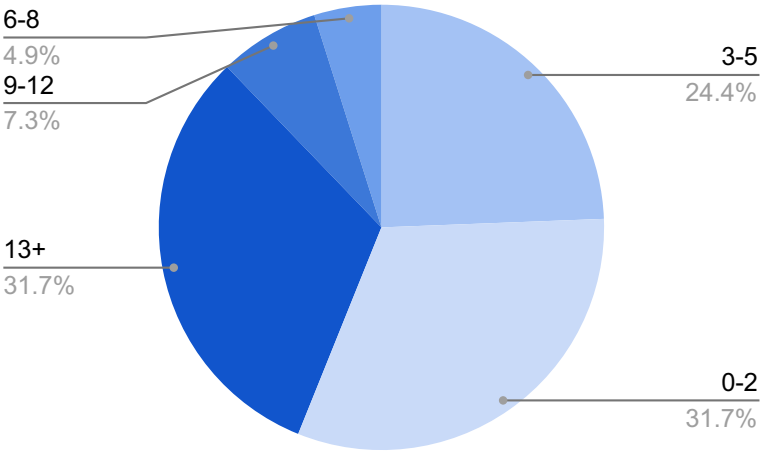
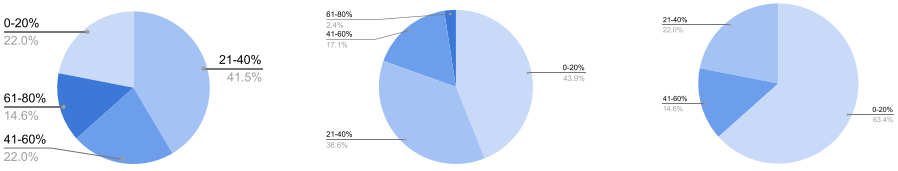


Fig. 17 Respondents' years of software engineering experience



(a) Writing code (b) Reading code (c) Reviewing code

Fig. 18 Percentages of respondents' daily work activities related to coding

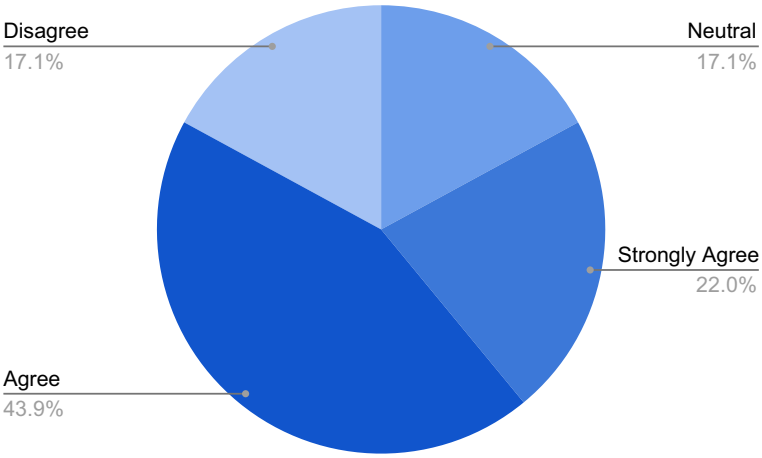


Fig. 19 Distribution of responses to the survey question: Do you agree that inline comments help you understand code better?

When we compared the results of our labeling process to the survey results, we saw some differences and similarities. We have encountered *obvious* comments the most, but according to practitioners this type of smell ranked lower, where the majority (73.1%) said they encounter such comments *rarely* or *never*. On the other hand, *task* comments were the most frequently encountered smell type according to practitioners with the highest weighted average. This smell type ranked as the 2^{nd} most encountered smell type during labeling.

While labeling the projects, we never encountered an *attribution* comment. Similarly, this type was the 2^{nd} least encountered smell type according to the practitioners. In addition, *irrelevant* comments ranked the lowest as the least encountered smell type by practitioners.

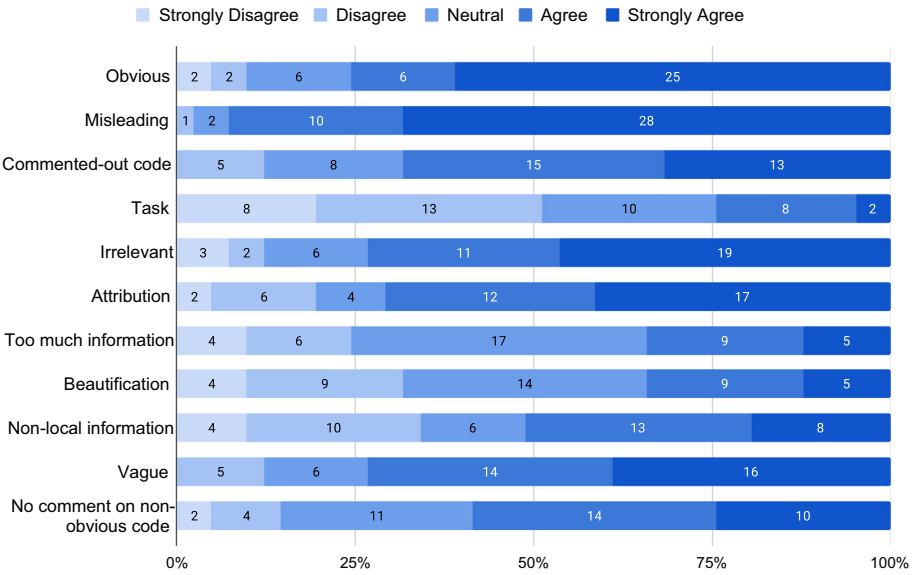


Fig. 20 Distribution of responses to the survey question: Do you agree that the given type of comment is a smell and should not be written in the source code?

Table 5 Statistics of the occurrences of the smell types observed by the survey respondents

Smell Type	Mean	SD	Median
Task	0.60	0.23	0.5
Commented-out code	0.59	0.22	0.75
Too much information	0.40	0.24	0.5
Vague	0.37	0.21	0.5
Beautification	0.35	0.25	0.25
Non-local info	0.32	0.25	0.25
Obvious	0.27	0.18	0.25
Misleading	0.26	0.18	0.25
Attribution	0.25	0.26	0.25
Irrelevant	0.08	0.16	0

The comparison of labeling and survey results suggests that *task*, *vague* and *beautification* comments occur more frequently than other smell types whereas *irrelevant* and *attribution* comments occur less frequently.

4.3.3 Perceptions on Impacts of Smells

Figure 22 shows the distribution of responses to the survey question about the effects of different smell types on software. In general, survey respondents found *misleading* comments negatively affecting software maintainability or code comprehension the most, as indicated by 90% of the respondents. The practitioners also found *irrelevant*, *vague*, and *no comment on non-obvious code* smells to have negative effects on software. On the other hand, *task*, *beautification*, *too much information*, and *non-local information* types of comments could

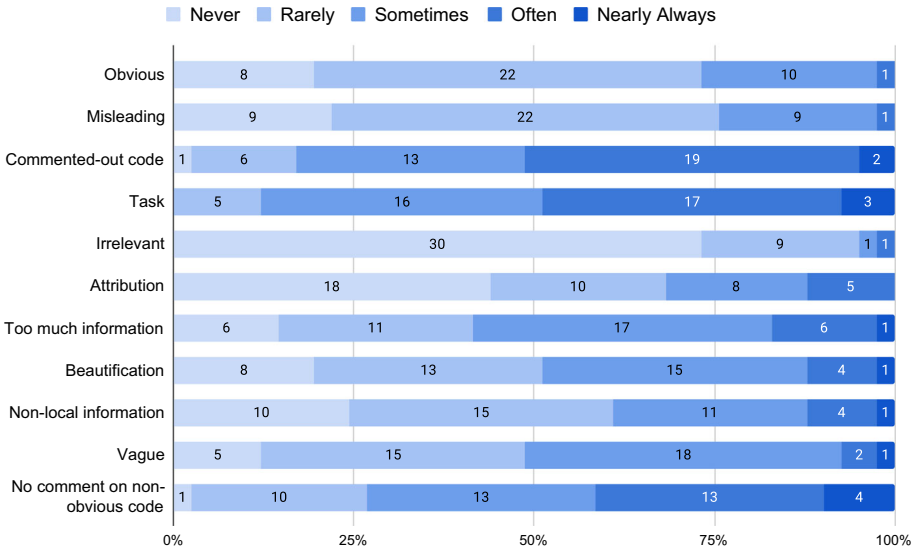


Fig. 21 Distribution of responses to the survey question: How often do you encounter this type of comment?

Table 6 Comparison of smell ranks obtained from the labeling process and practitioners' survey

Rank	Labeling (Total Number of Smells)	Survey (Weighted Average of Survey Results)
1	Obvious (758)	Task (0.60)
2	Task (130)	Commented-out code (0.59)
3	Vague (63)	Too much information (0.40)
4	Beautification (50)	Vague (0.37)
5	Commented-out code (38)	Beautification (0.35)
6	Misleading (19)	Non-local information (0.32)
7	Too much information (6)	Obvious (0.27)
8	Irrelevant (4)	Misleading (0.26)
9	Non-local information (4)	Attribution (0.25)
10	Attribution (0)	Irrelevant (0.08)

positively impact software as argued by the practitioners. The other types of smells were found to have negligible effects.

Summary of RQ3: What is the perception of software practitioners about inline code comment smells?

We surveyed 41 practitioners with varying levels of software engineering experience. They mostly agreed with the smell types we introduced except *task* comments, where some responded that they could actually be helpful. The respondents also mentioned that the *misleading* types of comments are the most harmful to software projects.

4.4 Acceptance of Pull Requests and Issues

In total, we opened 40 pull requests and 13 issues for the projects. Some developers accepted the changes as it is, some were rejected due to keeping task comments in the code or not accepting minor changes to the code and some did not receive any feedback. Since we opened both a pull request and an issue for task smells, we counted the smell being fixed if either the pull request or the issue was approved. Thus, in total, we had 45 fix attempts (by combining eight task smells fixes). Out of these 45, we had 11 approved, 4 ignored, and 30 rejected pull requests or issues. Our acceptance rate was 27% based on approved fixes over the approved and rejected fixes (excluding ignored cases).

The average response rate of the project maintainers' feedback was around 4.5 days. In *Kivy* project, three of the requests were rejected almost two months after we created them, which has a great effect on the response rate. The rate without including those three requests is around 9 hours. The responses we received from the project maintainers can be summarized as follows:

- Scikit-learn rejected all PRs and issues without reasoning.
- Requests project accepted only one issue. They provided an explanation for an issue regarding a vague smell. They rejected the other PRs as developers replied saying that they do not accept minor changes.

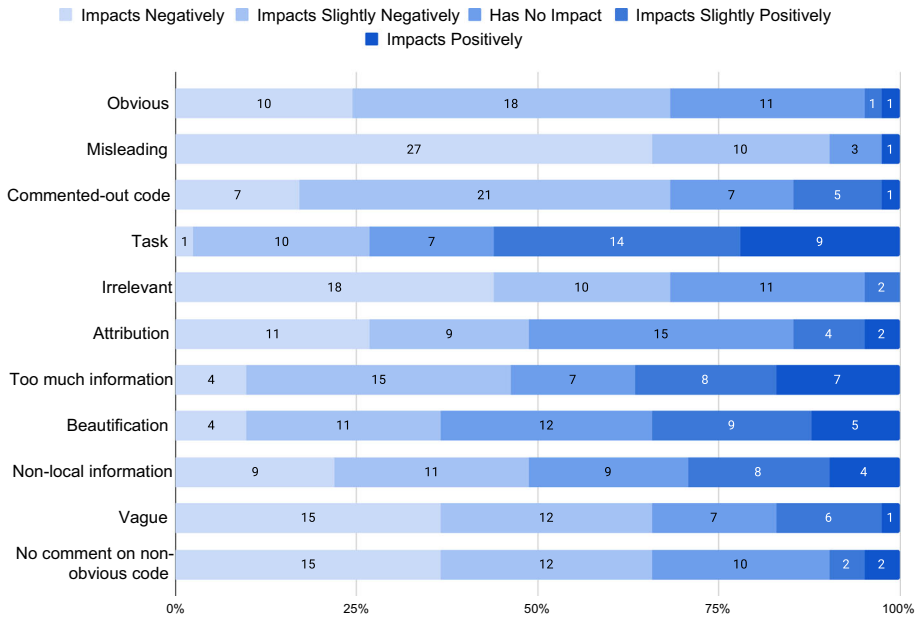


Fig. 22 Distribution of responses to the survey question: In your opinion, how does this comment type impact the software maintainability/code comprehension?

- The eight requests we submitted for the Light-4j project were all accepted. In one of the issues for this project, we pointed out a vague comment and asked them to provide more detail. They explained that the comment was in fact misleading and removed it later on.
- Moshi project accepted to remove an obvious comment but rejected the issues on task comments as they preferred the keep their tasks in the code.
- Jitsi project accepted to remove a comment that was misleading. However, they also rejected task comments as they preferred to have them in the code.
- Anki-Android project either ignored or rejected the PRs we submitted to the project with no explanation.
- Kivy project ignored three and rejected three of the six PRs we submitted.
- Scrapy project rejected all of the PRs and issues for no reason except for one, as they found another related issue in our PR that suggested removing a task comment.

Overall, our PR acceptance rate was 27%. The number of accepted, ignored, and rejected pull requests/issues is given in Fig. 23.

Summary of RQ4: Do developers of open-source projects accept pull requests or issues to remove inline comments smells?

To get the developers' reactions to fixing the comment smells in their projects, we opened 53 pull requests and issues in the GitHub repository of the selected projects and achieved a 27% acceptance rate. Some of the pull requests got rejected without any reason and some were due to the nonacceptance of minor changes in their project.

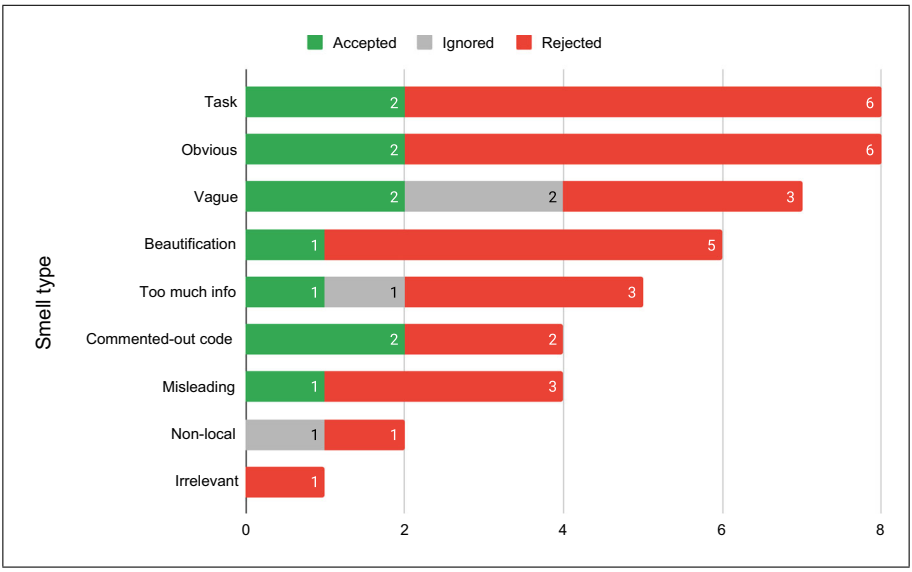


Fig. 23 Acceptance of pull requests and issues for each smell type

5 Discussion

This section discusses the potential reasons behind the results we obtained from labeling comment smells and the survey, as well as gives implications for researchers, practitioners, and educators.

5.1 Revisiting Research Questions

We revisit each of the research questions that we defined in the Introduction section to further discuss our findings and their possible explanations.

5.1.1 RQ1. What Types Of Inline Comment Smells Are There?

We have identified 11 inline comment smell types as a result of a multivocal literature review and manual analysis done on open-source Java and Python projects. We also got practitioners' points of view on these smells through a survey. Although the practitioners generally agreed with the taxonomy, there could be cases where comments we defined as smells might not be considered a smell in certain circumstances. Practitioners indicated that *task* comments might be helpful to grab the attention of developers. For *beautification* smell, they pointed out that these comments can ease reading a code. We also encountered supportive guidelines about writing *task* and *beautification* comments in the Anki-Android project. Survey respondents also mentioned that *too much information* and *the comment giving non-local information* might help newcomers understand the project better.

Since most of the survey respondents agreed with our smell taxonomy (except *task* comments), we included those types in our final smell taxonomy. For *task* comments, because we got a reasonable amount of responses saying they are generally forgotten, we also considered it as a smell and added it to our final taxonomy.

Although we have conducted an MLR, analyzed Java and Python projects, and carried out a practitioner survey, there might be additional smell types we are unaware of in these languages as well as in other languages. More specifically, different languages might require different commenting practices depending on their ability to express (Koornhof 2015). Considering that the languages we manually analyzed are easy to read, the possibility of different smell types occurring in languages that have lower expressiveness should not be overlooked. We should also mention that our taxonomy is based on the semantics of comments. Structural smells may also exist in projects with guidelines on comment structure. For instance, PEP 8 guidelines state that the first letter of the comment should be capitalized, there should be two spaces after sentence-ending periods in multi-sentence comments, etc.

We have encountered various types of comments that could be considered smells; however, we decided to label them as *not a smell* due to our labeling procedure. One such case is an inline comment restating the documentation comment (duplication). Because we ignored documentation comments from labeling, such comments were labeled as *not a smell*. Duplicate comments may lead to code-comment inconsistencies, as indicated in the study of Blasi et al. (2021). In Python, some comments referred to what type of object should be assigned to a variable. Although we labeled them as *not a smell*, they could be considered a smell if a project requires type hinting. Comments referring to common language keywords (e.g., __name__ in Python) could be considered as *obvious* smells since developers with enough experience would know them, and they could be easily found by an online search. However, we labeled them as *not a smell* since they might be helpful to novices.

5.1.2 RQ2. How Often Does Each Smell Type Occur In Practice?

There can be various reasons why a smell type occurs frequently or rarely in practice. We came up with several possible reasons to explain this:

- During labeling, we did not encounter any *attribution* comments in any of the projects. We encountered all the other smell types at least in two projects. This might be because version control systems such as GitHub are being used, and there is no longer a need to have *attribution* comments.
- We inspected the comment guidelines of the projects, which might have an impact on which smell type occurs more or less. We have observed that Anki-Android has the highest percentage of *task* and *beautification* comments. This might have been caused by its comment guidelines regarding these smell types, which encourage contributors to use them. Jitsi's code conventions regarding *commented-out code* might have led to Jitsi having the highest percentage for this smell type. *Scikit-learn*'s guidelines imply that *obvious* or *vague* comments should be pointed out during code review, which might be the reason why it has lower percentages of these smell types. Although the Python projects included guidelines on avoiding specific types of comments, we have observed that developers did not comply with them fully. The reason may be related to developers avoiding comment guidelines, which was also observed in the study of Rani et al. (2021) on comment styles.
- We have observed that all four Python projects have fewer percentages of comment smells than all four Java projects. This might be due to the number of stars and contributors of the projects. We observed that all Python projects have a higher number of stars and a higher number of contributors than all Java projects, which might be the reason why we encountered fewer smells in Python projects. Additionally, we also found out that the guidelines of all four Python projects include statements regarding avoiding *obvious*,

vague, and *misleading* comments in contrast to guidelines of all four Java projects that do not (see Table 7). Having guidelines that encourage avoiding these types of comment smells might also be another reason why we encountered fewer smells when we labeled Python projects. Since Python and Java are two different programming languages, language pragmatics might also have played a role in these results. However, to confirm this statement further research and studies of the two languages should be conducted.

5.1.3 RQ3. What is the Perception Of Software Practitioners About Inline Comment Smells?

Our survey results indicate that there are differing opinions on whether a comment type should be considered a smell and how often this smell type is encountered. This can be due to several reasons:

- Practitioners who completed our survey had varying levels of software engineering experience. This difference might have affected the practitioners' views on the smells. For example, a practitioner with 13+ years of experience is more likely to encounter a certain type of smell than a practitioner with one year of experience.
- Practitioners are more likely to work in different sectors and workplaces. Considering that each workplace and sector can come with different work cultures, this might affect their answers in the survey. For example, *task* and *beautification* comments received an almost equal number of agreements and disagreements regarding whether they should be counted as smells. Some workplaces might be encouraging these types of comments, whereas others might not. Hence, practitioners have different views regarding these smell types.
- All eight projects we have labeled are open-source projects. Practitioners who participated in our survey do not have to necessarily work with open-source projects, so this might have caused differences in the survey and empirical results. For example, the most encountered smell type is *commented-out code* according to the practitioners. Our empirical research showed that *obvious* comments are the most common smell type in the labeled projects.

5.1.4 RQ4. Do Developers Of Open-Source Projects Accept Pull Requests Or Issues To Remove Inline Comments Smells?

Due to the manual submission of pull requests and issues, we could not submit hundreds of them. In addition, we observed that developers respond negatively to the high number of requests that deal with similar issues as they see them as spam. Hence, we chose to attempt fixing only at most one comment from each smell type for each project. We attained an acceptance rate of 27%. On average, the acceptance of pull requests to open-source projects is 29% Medeiros et al. (2019). Although it is low, we obtained a very close result to the average.

Our results indicate that the acceptance of removing inline comment smells heavily relies on the preferences of individuals and teams. In total, we had 11 accepted pull requests or issues. 7 of them were accepted by the developers of Light-4j. This suggests that some individuals and teams are more welcoming toward these types of requests that remove an inline comment smell. 30 of the requests we opened were rejected. The fact that four of them (13.5%) were rejected due to developers preferring to maintain task comments in the code base highlights that some teams may prioritize the inclusion of tasks in the code base

Table 7 Projects with contribution guidelines mentioning which types of smells should be avoided (asterisks refer to indirect references)

	Java				Python			
	Anki-Android	Jitsi	Moshi	Light-4j	Requests	Scrapy	Kivy	Scikit-learn
Misleading	–	–	–	–	✓*	✓*	✓*	✓*
Obvious	–	–	–	–	✓*	✓*	✓*	✓
No comment on non-obvious code	–	–	–	–	–	–	–	✓
Commented-out code	–	–	–	–	–	–	–	
Irrelevant	–	–	–	–	–	–	–	✓
Task	–	–	–	–	–		–	
Too much information	–	–	–	–	–	–	–	–
Attribution	–	–	–	–	–	–	–	–
Beautification	–	–	–	–	–	–	–	–
Non-local	–	–	–	–	–	–	–	–
Vague	–	–	–	–	–✓*	✓*	✓*	✓

over the potential threats of inline comment smells. Furthermore, 8 requests were rejected (26.5%) due to developers not accepting minor changes. This indicates that some teams may be resistant to changing code that is already working even if removing the comment smell would be beneficial but a minor adjustment. Lastly, 18 requests were rejected (60%) as the developers did not provide any information regarding their decision. It is likely that they saw our PRs as spam or decided to not consider them since the PRs were not fixing a bug or adding a new functionality.

Two of the projects that we submitted requests to explained that they prefer keeping task comments in the code base. We observed a similar case in our survey where some developers mentioned that they do not consider task comments as smells. The results indicate that the definition of comment smells of programmers may not align with that of the literature due to the preferences of individuals and teams. However, according to the feedback to our requests and survey results, the comment smell types that are most likely to receive inconsistent opinions are task and beautification smells. Thus, the alignment of definitions of comments smells among programmers and in the literature can change depending on the smell types.

In general, the acceptance of removing inline comment smells is largely determined by the preferences of individuals and teams, which may be the reason for the low acceptance rate that we obtained. Although removing comment smells is inherently valuable for the code base, developers of open-source projects generally do not prefer pull requests that propose such a change. Thus, our acceptance rate indicates that removing comment smells can be beneficial for the code base though it is not a priority for the developers.

5.2 Implications for Researchers

There are various ways our findings can be utilized by researchers. In this subsection, we explain how and what type of research can be conducted about comment smells in relation to our findings.

- We have established a taxonomy of comment smells and labeled 2447 comments. The dataset we published can be referenced by researchers to develop tools that can be used to detect comment smells and identify their types, which will contribute to increasing the readability and writability of code. The taxonomy and labeling of the comments can be utilized as a reference point to identify comment smells that occur. Understanding which types of comment smells occur the most and why can enhance the comment quality of code.
- The eight manually labeled projects of two different languages, Java and Python, imply that there are differences in the number of smells they have. It is recommended that researchers conduct similar labeling processes for different languages to see if there are differences in the number of smells in correlation to the language. This can improve our understanding of comment smells in terms of why they occur and how their numbers can be decreased. Additionally, through the labeling of more projects, the taxonomy can be extended to include more types.
- How our understanding of this taxonomy of comment smells can play a role in comment quality is a question that is still unanswered. Case studies can be conducted to answer this question. Software engineers who have been exposed and not exposed to the taxonomy can be compared in terms of the number of comment smells they can identify or have in their code.
- We have conducted a survey for practitioners to validate our taxonomy. We observed that some participants did not view some of the types in our taxonomy as smells. For example,

task and *beautification* comments improved the quality of the code according to some practitioners. Hence, approaches from people with different backgrounds or experience levels in the software industry can be further investigated.
– Comment smells can occur at different stages of the software development cycle. For example, smells can be overlooked during code reviews. These stages can be inspected to identify which smells occur at which stages. Acquiring this knowledge can lead to finding new ways of avoiding comment smells.

5.3 Implications for Practitioners

We observed that out of eight projects, only three of them had guidelines regarding how the comments should be written. These guidelines only hinted at one to three smell types identified in this paper and did not cover writing comments as detailed as in other sections. This suggests that projects have a low number of comment guidelines.

Although these guidelines regarding the comments are relatively limited in terms of the smell types they mention, the observations imply that guidelines on comments can have an impact on the comment quality, as we have discussed in the previous section. Hence, it can be recommended that practitioners include comment guidelines in their projects to increase comment readability and writability.

Furthermore, a model can be implemented by practitioners to detect inline comment smells automatically in their projects. This would facilitate the reviewing process and help code reviewers detect comment smells. Ultimately, this would lead to an increase in the quality of the comments on the projects.

5.4 Implications for Educators

In the software engineering education curriculum, there is a lack of training in code comment writing. The proper way of using inline comments can be overlooked in the lectures.

We tried to find lecture notes regarding code comments by Google Search, which yielded results that were scarce in number. Even though we found sources that give guidelines for comments, most of them lacked detailed information regarding distinct comment and smell types. These sources include lecture notes from Stanford University (Ousterhout 2015), University of Utah (Commenting 2023), and University of Puget Sound (Chun et al. 2023) that encourage commenting on non-obvious code and avoiding *obvious* comments.

In introductory courses, instructors often encourage students to write too many comments, to the extent that their projects include many *obvious* comments that explain basic code. For example, a source from the University of Washington (of Washington 2023), specifically gives a definition for inline comments in contrast to the above-mentioned sources. However, this source does not give a direct guideline for writing an inline comment and even gives an example that can be labeled as an *obvious* comment. This inline comment example and the lack of guidelines can lead a student to explain obvious code, which causes the number of smells to increase in code.

This claim can further be supported by blog entries of two senior software engineers (Spertus 2021; Kunk 2011). Both blog entries agree that students feel like they must write a lot of comments for a good grade. Without the required training in comment writing, this can lead many students write comments for the sake of writing them. While this might be beneficial for extreme beginners to understand coding, it might cause a habit of explaining obvious code.

The taxonomy of comment smells can be introduced in lectures to software engineering candidates. Educators can use our study to organize a session that focuses on comment smells and their importance on code comprehension. This can raise awareness of code readability and writability, which can lead to projects having comments of better quality.

6 Threats to Validity

In the following, we discuss potential threats to the internal and external validity of the study.

Internal Validity: There are various validity threats to the design of our study. First of all, our search query defined for MLR may not capture some works on comment smells, and we may miss important sources. We mitigated this threat by deciding the keywords based on previous similar MLR studies. Execution of the defined query in Google Search (for grey literature) was conducted by two authors for the mitigation of the filter bubble. Secondly, we analyzed the sources provided by Google Search and Google Scholar. There could be academic or gray literature materials related to the study that were not indexed by these engines. To minimize the risk of any subjectivity in the MLR process, we followed the MLR guidelines of Garousi et al. (2019). During the primary study selection and data extraction, all steps were completed by two of the authors independently. When there was any conflict, the third author was consulted in the decision process.

The documentation comments in Java were filtered out by checking whether a given comment starts with "/**". However, in some cases, there were method-level comments starting with "//" or "/*", which were filtered out manually during the labeling process and replaced with inline comments. We also ignored the license and linter-disabling (e.g., #noqa in Python) comments during the labeling process. Thus, the number of inline comments shown in Table 2 is not exact.

We sampled the comments with file granularity, which led to 266 comments being duplicates due to the same piece of code being written in multiple places in a source file (especially in test codes). This could potentially create a bias in the analysis. However, we believe that file-level sampling helped us to label the comments with higher confidence.

Although two authors labeled the comments, there could still be a possibility of mislabeled comments. For instance, annotating a comment as *misleading* requires understanding the related code part completely. Moreover, distinguishing *vague* and *non-local* comments from *not a smell* was sometimes difficult as they required a decent amount of domain knowledge.

Furthermore, there are several threats to the validity of the survey results. To give an incentive for participation, we promised $100 Amazon Gift card to one of the participants who would be chosen randomly. Even though this helped us recruit developers from different backgrounds, it might also have created a sampling bias in the survey by attracting people who are completing it for its prize.

The survey had 52 questions that included multiple choice and open-ended questions, which are estimated to take 15-20 minutes to complete. The length of the survey might also cause biased answers because the attention span of a participant decreases after spending 10 minutes on a survey (Sen 2023). This increases the likelihood of a participant answering without thinking carefully or picking a random response.

Another potential threat to the survey is that the respondents may misunderstand the smell definitions. To mitigate this issue, we provided a description and an accompanying example for each smell type. We also conducted a pilot test with five software engineers from the authors' network before sending the survey to the practitioners to make sure that the questions were understandable. We adjusted the questions according to their feedback to make sure that the questions and smells presented in the survey were comprehensive.

External Validity: An external validity threat related to the analysis of only eight projects is the limited generalizability of the study's findings to other software projects or contexts. Because the sample size is small and may not be representative of the broader population of software projects, the results obtained from these eight projects may not be applicable to other projects with different characteristics, development processes, team structures, or technologies.

Lastly, we submitted 53 pull requests and/or issues, which is relatively small to make a general statement about the developers' perception of removing code smells from open-source projects. Moreover, the pull requests were open to the personal opinions of the developers who are reviewing them as we encountered such cases. All developers have different priorities regarding the code; thus, it is difficult to generalize the results to all developers of open-source projects.

7 Conclusion and Future Work

Code comments are valuable items when it comes to documenting and explaining source code. Having low-quality comments negatively affects source code comprehension and deteriorates software quality. In this study, we present a taxonomy of inline code comment smells consisting of 11 categories. Our manual analysis of eight open-source Java and Python projects revealed that inline comment smells exist in projects, and the majority of them are *obvious* comments. We also published the manually labeled smell dataset online. Moreover, we conducted a survey with software practitioners to get their opinions on the inline comment smells and their effects on program comprehension and maintainability. The majority of respondents agreed with the smells defined in our taxonomy except for *task* comments and they found *misleading* comments to be the smell type that negatively affects code comprehension and maintainability the most. We submitted 53 pull requests and/or issues to remove comment smells from open-source projects in order to better understand their significance for developers, where we achieved an acceptance rate of 27%. Despite the indication that removing comment smells is beneficial, we received a low acceptance rate as developers of open-source projects tend to reject pull requests that do not necessarily fix issues and oppose changing working source code.

There are several implications of the study for software practitioners, researchers, and educators. First, researchers can build automated models to recognize comment smells in a project using our published dataset. Second, our taxonomy can be used as a baseline to be extended by researchers. Third, practitioners can include our smell taxonomy in their project contribution guidelines on code comments. Lastly, educators can raise awareness of code comprehension by having lectures about our smell taxonomy.

In future work, we plan to develop a model for automatically detecting smell types by utilizing natural language processing and machine learning techniques. Another challenge that we would also like to undertake is to investigate *no comment on non-obvious code* smell type and develop algorithms to find the occurrences of that smell automatically.

Appendix

The list of sources we gathered from grey and white literature during MLR is given in Tables 8 and 9.

Table 8 Summary of MLR results (part 1)

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Clean code: A handbook of agile software craftsmanship Martin (2009)	Gray	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Putting comments in code: the good, the bad, and the ugly. Sourour (2017)	Gray		✓							✓	
How to Comment Your Code Like a Pro: Best Practices and Good Habits Keeton (2019)	Gray				✓					✓	
What Makes a Good Code Comment? Cronin (2019)	Gray		✓								
Commenting Commenting (2023)	Gray		✓								
Code Comment Is A Smell Brack (2016)	Gray						✓	✓			
Don't comment your code, Rewrite it! Trivedi (2019)	Gray	✓	✓		✓						
Is There a Correct Way to Comment Your Code? Dietrich (2017)	Gray	✓						✓			
Writing Comments in Python Zhané (2023)	Gray		✓					✓			
Commenting Python Code Hofmann (2023)	Gray		✓					✓			
No Comment: Why Commenting Code Is Still a Bad Idea Vogel (2013)	Gray		✓					✓			

Table 8 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
How to Write a Good Code Comment Adams (2020)	Gray		✓					✓			
Useless comments can ruin your code reviews. Here's how to erase them Troisi (2023)	Gray		✓		✓						
The Hows and Whys of Commenting C and C++ Code Allain (2023)	Gray		✓					✓			
Stop Writing Code Comments Norlander (2019)	Gray	✓	✓		✓			✓			✓
Clean Code - Code versus Comments Molloy (2020)	Gray	✓	✓								
Best practices on how to write comments in your code Lelli (2019)	Gray		✓					✓			
Why We Comment Code Geiser (2017)	Gray	✓						✓			
Code Tells You How, Comments Tell You Why Atwood (2006)	Gray							✓			
Please, don't commit commented out code Dodds (2015)	Gray				✓						
Code Health: To Comment or Not to Comment? Reuveni and Bourrillion (2017)	Gray		✓					✓			

Table 8 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
5-best-practices-for-commenting-your-code Fuex (2011)	Gray		✓				✓		✓		
How To Write Code Comments Well Januska (2012)	Gray		✓						✓		
5 Code Commenting Don'ts Tozzi (2020)	Gray		✓			✓	✓				
Don't share commented-out code Nayuki (2018)	Gray					✓					
Learn which types of comments should be avoided Schults (2017)	Gray	✓	✓		✓		✓				
Don't comment your code! Woost (2011)	Gray	✓			✓				✓	✓	
Good comment, bad comment Marcus (2018)	Gray	✓	✓	✓							
7 ways to write bad comments Hilton (2014)	Gray	✓	✓	✓							
To comment or not to comment? // that is the question Boccara (2017)	Gray	✓	✓								
Code Comments are Lies Koornhof (2015)	Gray	✓	✓								✓
Good Comments and Bad Comments in Your Program Heartin (2018)	Gray	✓	✓	✓	✓		✓		✓		
Are Code Comments A Code Smell? Are code comments a code smell (2018)	Gray							✓			

Table 8 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Writing code comments Writing code comments (2023)	Gray		✓					✓			
What Are the Biggest Code Commenting Fails You've Seen? Fronczak (2018)	Gray		✓				✓		✓	✓	
I'll Delete Your Commented Code Without Reading It and I'm Not Sorry McEwen (2018)	Gray				✓						
4 Reasons Why We Need Code Comments Henke (2018)	Gray							✓			
Commented Out Code Is Junk In Your Codebase Morlion (2017)	Gray				✓						
Why Good Codes Don't Need Comments Nguyen (2020)	Gray		✓							✓	✓
To Comment or Not to Comment Kunk (2011)	Gray	✓								✓	
Improve code readability by getting rid of comments Krasnov (2020)	Gray	✓	✓								
How to Write Comments the Right Way Parashar (2020)	Gray		✓								
Coding and Comment Style Larson (2023)	Gray	✓	✓								

Table 9 Summary of MLR results (part 2)

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
10 Best Practices to Follow While Writing Code Comments Paul (2021)	Gray		✓	✓							
13 Tips to Comment Your Code Aguilar (2008)	Gray	✓	✓								
To comment or not to comment? McDonald (2019)	Gray	✓	✓		✓						
Fighting Evil in Your Code: Comments on Comments Sorens (2017)	Gray	✓	✓		✓					✓	✓
9 Tips That Promote Clean Code: Writing Comments in a Good way Herath (2020)	Gray		✓		✓						
The pros and cons (but mostly pros) of comments in code Asay (2019)	Gray	✓									
Best practices for writing code comments Spertus (2021)	Gray	✓									
To write code comments or not, it should not be a question Struyf (2021)	Gray	✓		✓							
When Should You Add Comments to Your Code? Saha (2023)	Gray		✓	✓				✓		✓	
3 kinds of good comments Hilton (2014)	Gray							✓			

Table 9 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Code Review: Please Remove Comments From Your Code Huy (2021)	Gray	✓						✓			
Quality Analysis of Source Code Comments Steidl et al. (2013)	White	✓	✓		✓			✓			
Automatic Quality Assessment of Source Code Comments: The JavadocMiner Khamis et al. (2010)	White	✓									
Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes Fluri et al. (2007)	White	✓						✓			
@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies Tan et al. (2012)	White	✓									
/* iComment: Bugs or Bad Comments? */ Tan et al. (2007)	White	✓						✓			
Analyzing Code Comments to Boost Program Comprehension Shinyama et al. (2018)	White							✓			

Table 9 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Learning to Update Natural Language Comments Based on Code Changes Panthaplackel et al. (2020)	White	✓									
A Large-Scale Empirical Study on Code-Comment Inconsistencies Wen et al. (2019)	White	✓									
Comments Are More Important Than Code Raskin (2005)	White	✓	✓					✓			
HotComments: How to Make Program Comments More Useful? Tan et al. (2007)	White	✓									
Learning Code Context Information to Predict Comment Locations Huang et al. (2019)	White					✓		✓			
Where should I comment my code? A dataset and model for predicting locations that need comments Louis et al. (2020)	White	✓									
Towards Detecting Inconsistent Comments in Java Source Code Automatically Stulova et al. (2020)	White	✓									

Table 9 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Study of a code comment decision method based on structural features Wang et al. (2019)	White				✓						✓
AutoComment: Mining Question and Answer Sites for Automatic Comment Generation Wong et al. (2013)	White		✓								
Deep Code-Comment Understanding and Assessment Wang et al. (2019)	White	✓								✓	
Understanding the Rationale for Updating a Function's Comment Malik et al. (2008)	White	✓									
How Good is your Comment? A study of Comments in Java Programs Haouari et al. (2011)	White	✓			✓						
Automatically Detecting the Up-To-Date Status of ToDo Comments in Java Programs Sridhara (2016)	White										✓
Detecting Code Comment Inconsistency using Siamese Recurrent Network Rabbi and Siddik (2020)	White	✓									✓

Table 9 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers Storey et al. (2008)	White										✓
Examining the Evolution of Code Comments in PostgreSQL Jiang and Hassan (2006)	White	✓									
Listening to Programmers - Taxonomies and Characteristics of Comments in Operating System Code Padioleau et al. (2009)	White										✓
Detecting Fragile Comments Ratol and Robillard (2017)	White	✓									
Deep Just-In-Time Inconsistency Detection Between Comments and Source Code Panthaplackel et al. (2021)	White	✓									
Automating Just-In-Time Comment Updating Liu et al. (2020)	White	✓									
A Topic Modeling Approach To Evaluate The Comments Consistency To Source Code Iammarino et al. (2020)	White	✓									

Table 9 continued

Source	Type	Misleading	Obvious	Too much information	Commented out code	Nonlocal information	Attribution	No comment on non-obvious code	Beautification	Irrelevant	Task
Deep Learning to Detect Redundant Method Comments Louis et al. (2018)	White	✓	✓								
CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis Zhai et al. (2020)	White	✓									

Funding Open access funding provided by the Scientific and Technological Research Council of Türkiye (TÜBİTAK).

Data Availability The labeled inline comment smells dataset and MLR sources are available at https://doi.org/10.6084/m9.figshare.19640886.v9.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

References

Commenting (2023) https://www.cs.utah.edu/~germain/PPS/Topics/commenting.html. Accessed 10-04-2022

Writing code comments (2023) https://web.archive.org/web/20210327171829/https://developers.google.com/tech-writing/two/code-comments. Accessed 28-04-2022

Are code comments a code smell? (2018). https://steemit.com/programming/@beggars/are-code-comments-a-code-smell. Accessed 28-04-2022

Adams T (2020) How to write a good code comment. https://www.pullrequest.com/blog/how-to-write-a-good-code-comment/. Accessed 28-04-2022

Aguilar JM (2008) 13 tips to comment your code. https://www.devtopics.com/13-tips-to-comment-your-code/. Accessed 28-04-2022

Allain A (2023) The hows and whys of commenting c and c++ code. https://www.cprogramming.com/tutorial/comments.html. Accessed 28-04-2022

Allamanis M, Tarlow D, Gordon A, Wei Y (2015) Bimodal modelling of source code and natural language. In: International conference on machine learning, PMLR pp 2123–2132

Asay M (2019) The pros and cons (but mostly pros) of comments in code. https://www.techrepublic.com/article/the-pros-and-cons-but-mostly-pros-of-comments-in-code/. Accessed 28-04-2022

Atwood J (2006) Code tells you how, comments tell you why. https://blog.codinghorror.com/code-tells-you-how-comments-tell-you-why/. Accessed 28-04-2022

Blasi A, Goffi A, Kuznetsov K, Gorla A, Ernst MD, Pezzè M, Castellanos SD (2018) Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 242–253

Blasi A, Stulova N, Gorla A, Nierstrasz O (2021) Replicomment: identifying clones in code comments. J Syst Softw 182:111069

Boccara J (2017) To comment or not to comment? // that is the question. https://www.fluentcpp.com/2017/05/02/to-comment-or-not-to-comment-that-is-the-question/. Accessed 28-04-2022

Boswell D, Foucher T (2011) The Art of Readable Code: Simple and Practical Techniques for Writing Better Code. O'Reilly Media. https://books.google.com.tr/books?id=RPryrfu1iP4C

Brack F (2016) Code comment is a smell. https://fagnerbrack.com/code-comment-is-a-smell-4e8d78b0415b. Accessed 28-04-2022

Chun C, O'Neil K, Young K, Christoph JN (2023) Writing code and code comments. https://soundwriting.pugetsound.edu/pugetsound/subsection-code.html. Accessed 03-10-2022

Cochran WG (2007) Sampling techniques. John Wiley & Sons

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Measur 20(1):37–46

Corazza A, Maggio V, Scanniello G (2018) Coherence of comments and method implementations: a dataset and an empirical investigation. Software Qual J 26(2):751–777

Cronin M (2019) What makes a good code comment? https://itnext.io/what-makes-a-good-code-comment-5267debd2c24. Accessed 28-04-2022
Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. In: NAACL
Dietrich E (2017) Is there a correct way to comment your code? https://blog.ndepend.com/correct-way-comment-code/. Accessed 28-04-2022
Dodds KC (2015) Please, don't commit commented out code. https://kentcdodds.com/blog/please-dont-commit-commented-out-code. Accessed 28-04-2022
Doğan E, Tüzün E (2022) Towards a taxonomy of code review smells. Inf Softw Technol 142:106737
Fluri B, Wursch M, Gall HC (2007) 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 pp 70–79
Fronczak S (2018) What are the biggest code commenting fails you've seen? https://blog.submain.com/biggest-code-commenting-fails/. Accessed 28-04-2022
Fuex J (2011) 5 best practices for commenting your code. https://improvingsoftware.com/2011/06/27/5-best-practices-for-commenting-your-code/. Accessed 28-04-2022
Garousi V, Felderer M, Mäntylä MV (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf Softw Technol 106:101–121
Garousi V, Küçük B (2018) Smells in software test code: A survey of knowledge in industry and academia. J Syst Softw 138:2–81
Geiser M (2017) Why we comment code. https://dzone.com/articles/why-we-comment-code-yet-another-code-commenting-ar. Accessed 28-04-2022
Godin K, Stapleton J, Kirkpatrick SI, Hanning RM, Leatherdale ST (2015) Applying systematic review search methods to the grey literature: a case study examining guidelines for school-based breakfast programs in canada. Syst Rev 4(1):1–10
Goffi A, Gorla A, Ernst MD, Pezzè M (2016) Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th international symposium on software testing and analysis, pp 213–224
Gvero T, Kuncak V (2015) Synthesizing java expressions from free-form queries. In: Proceedings of the 2015 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications, pp 416–432
Haouari D, Sahraoui H, Langlais P (2011) How good is your comment? a study of comments in java programs. In: 2011 International symposium on empirical software engineering and measurement, IEEE pp 137–146
Hartzman CS, Austin CF (1993) Maintenance productivity: Observations based on an experience in a large system environment. In: Proceedings of the 1993 conference of the centre for advanced studies on collaborative research: software engineering-vol 1, pp 138–170
Heartin (2015) Good comments and bad comments in your program. https://www.javajee.com/good-comments-and-bad-comments-in-your-program. Accessed 28-04-2022
Henke M (2018) 4 reasons why we need code comments. https://blog.submain.com/4-reasons-need-code-comments/. Accessed 28-04-2022
Herath P (2020) 9 tips that promote clean code: Writing comments in a good way. https://javascript.plainenglish.io/clean-code-writing-comments-in-a-good-way-8203c7d80c65/. Accessed 28-04-2022
Hilton P (2014) 3 kinds of good comments. https://hilton.org.uk/blog/3-kinds-of-good-comments. Accessed 28-04-2022
Hilton P (2014) 7 ways to write bad comments. https://hilton.org.uk/blog/7-ways-to-write-bad-comments. Accessed 28-04-2022
Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
Hofmann F (2023) Commenting python code. https://stackabuse.com/commenting-python-code/. Accessed 28-04-2022
Huang Y, Hu X, Jia N, Chen X, Xiong Y, Zheng Z (2019) Learning code context information to predict comment locations. IEEE Trans Reliab 69(1):88–105
Huy NVK (2021) Code review: Please remove comments from your code. https://levelup.gitconnected.com/code-review-please-remove-comments-from-your-code-2c4de8cf9b13. Accessed 28-04-2022
Iammarino M, Aversano L, Bernardi ML, Cimitile M (2020) A topic modeling approach to evaluate the comments consistency to source code. In: 2020 International joint conference on neural networks (IJCNN), IEEE pp 1–8
Jabrayilzade E, Gürkan O, Tüzün E (2021) Towards a taxonomy of inline code comment smells. In: 2021 IEEE 21st International working conference on source code analysis and manipulation (SCAM), IEEE pp 131–135
Januska A (2012) How to write code comments well. https://web.archive.org/web/20191223045018/https://antjanus.com/blog/daily-gibberish/best-comment-separator/. Accessed 28-04-2022

Jiang ZM, Hassan AE (2006) Examining the evolution of code comments in postgresql. In: Proceedings of the 2006 international workshop on mining software repositories, pp 179–180
Keele S, et al. (2007) Guidelines for performing systematic literature reviews in software engineering. Tech rep, Technical report, Ver 2.3 EBSE Technical Report. EBSE
Keeton B (2019) How to comment your code like a pro: Best practices and good habits. https://www.elegantthemes.com/blog/wordpress/how-to-comment-your-code-like-a-pro-best-practices-and-good-habits. Accessed 28-04-2022
Khamis N, Witte R, Rilling J (2010) Automatic quality assessment of source code comments: the javadocminer. In: International Conference on application of natural language to information systems, Springer pp 68–79
Khan JY, Khondaker MTI, Uddin G, Iqbal A (2021) Automatic detection of five API documentation smells: Practitioners' perspectives. In: 2021 IEEE International conference on software analysis, evolution and reengineering (SANER), IEEE pp 318–329
Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Software Eng 32(12):971–987
Koornhof P (2015) Code comments are lies. https://www.codeproject.com/Articles/872073/Code-Comments-are-Lies. Accessed 28-04-2022
Krasnov M (2020) Improve code readability by getting rid of comments. https://everyday.codes/best-practices/improve-code-readability-by-getting-rid-of-comments/. Accessed 28-04-2022
Kunk J (2011) To comment or not to comment. https://visualstudiomagazine.com/articles/2011/01/06/to-comment-or-not-to-comment.aspx. Accessed 10-04-2022
Kunk J (2011) To comment or not to comment. https://visualstudiomagazine.com/Kunk0211. Accessed 28-04-2022
Larson D (2023) Coding and comment style. https://mitcommlab.mit.edu/broad/commkit/coding-and-comment-style/. Accessed 28-04-2022
LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th international conference on software engineering, pp 492–501
Lelli F (2019) Best practices on how to write comments in your code. https://francescolelli.info/programming/best-practices-on-how-to-write-comments-in-your-code/. Accessed 28-04-2022
Liu Z, Xia X, Lo D, Yan M, Li S (2021) Just-in-time obsolete comment detection and update. IEEE Transactions on Software Engineering pp 1–1. https://doi.org/10.1109/TSE.2021.3138909
Liu Z, Xia X, Yan M, Li S (2020) Automating just-in-time comment updating. In: Proceedings of the 35th IEEE/ACM International conference on automated software engineering, pp 585–597
Louis A, Dash SK, Barr ET, Ernst MD, Sutton C (2020) Where should I comment my code? a dataset and model for predicting locations that need comments. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering: new ideas and emerging results, pp 21–24
Louis A, Dash SK, Barr ET, Sutton C (2018) Deep learning to detect redundant method comments. arXiv preprint arXiv:1806.04616
Malik H, Chowdhury I, Tsou HM, Jiang ZM, Hassan AE (2008)Understanding the rationale for updating function's comment. In: 2008 IEEE International conference on software maintenance, IEEE pp 167–176
Marcus R (2018) Good comment, bad comment. https://rmarcus.info/blog/2018/11/05/good-bad-comment.html. Accessed 28-04-2022
Martin RC (2009) Clean code: a handbook of agile software craftsmanship. Pearson Education
McDonald JC (2019) To comment or not to comment? https://dev.to/codemouse92/to-comment-or-not-to-comment-3f7h. Accessed 28-04-2022
McEwen M (2018) I'll delete your commented code without reading it and i'm not sorry. https://blog.submain.com/delete-commented-code-without-reading/. Accessed 28-04-2022
Medeiros F, Lima G, Amaral G, Apel S, Kästner C, Ribeiro M, Gheyi R (2019) An investigation of misunderstanding code patterns in c open-source software projects 24(4). https://doi.org/10.1007/s10664-018-9666-x. https://doi.org/10.1007/s10664-018-9666-x
Misra V, Reddy JSK, Chimalakonda S (2020) Is there a correlation between code comments and issues? an exploratory study. In: Proceedings of the 35th Annual ACM symposium on applied computing, pp 110–117
Molloy S (2020) Clean code - code versus comments. https://nebulaconsulting.co.uk/insights/code-over-comments/. Accessed 28-04-2022
Morlion P (2017) Commented out code is junk in your codebase. https://blog.submain.com/commented-out-code-junk-codebase/. Accessed 28-04-2022
Nayuki (2018) Don't share commented-out code. https://www.nayuki.io/page/dont-share-commented-out-code. Accessed 28-04-2022

Neuhaus C, Neuhaus E, Asher A, Wrede C (2006) The depth and breadth of google scholar: An empirical study. Portal Libraries and the Academy 6(2):127–141
Nguyen N (2020) Why good codes don't need comments. https://towardsdatascience.com/why-good-codes-dont-need-comments-92f58de19ad2. Accessed 28-04-2022
Nielebock S, Krolikowski D, Krüger J, Leich T, Ortmeier F (2019) Commenting source code: is it worth it for small programming tasks? Empir Softw Eng 24:1418–1457
Norlander B (2019) Stop writing code comments. https://medium.com/@bpnorlander/stop-writing-code-comments-28fef5272752. Accessed 28-04-2022
Ousterhout J (2015) Writing comments. https://web.stanford.edu/~ouster/cgi-bin/cs190-spring15/lecture.php?topic=comments. Accessed 10-04-2022
Padioleau Y, Tan L, Zhou Y (2009) Listening to programmers-taxonomies and characteristics of comments in operating system code. In: 2009 IEEE 31st international conference on software engineering, IEEE pp 331–341
Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language api descriptions. In: 2012 34th international conference on software engineering (ICSE), IEEE pp 815–825
Panthaplackel S, Li JJ, Gligorić M, Mooney RJ (2021) Deep just-in-time inconsistency detection between comments and source code. In: AAAI
Panthaplackel S, Nie P, Gligoric M, Li JJ, Mooney RJ (2020) Learning to update natural language comments based on code changes. arXiv preprint arXiv:2004.12169
Parashar A (2020) How to write comments the right way. https://levelup.gitconnected.com/how-to-write-comments-the-right-way-8d13b24804bd. Accessed 28-04-2022
Pascarella L, Bacchelli A (2017) Classifying code comments in java open-source software systems. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), IEEE pp 227–237
Paul J (2021) 10 best practices to follow while writing code comments. https://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html#axzz7RlqtwW2w. Accessed 28-04-2022
Qamar KA, Sülün E, Tüzün E (2022) Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis. Inf Softw Technol 150:106972. https://doi.org/10.1016/j.infsof.2022.106972, www.sciencedirect.com/science/article/pii/S0950584922001094
Rabbi F, Siddik MS (2020) Detecting code comment inconsistency using siamese recurrent network. In: Proceedings of the 28th international conference on program comprehension, pp 371–375
Rani P, Abukar S, Stulova N, Bergel A, Nierstrasz O (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 pp 165–169
Raskin J (2005) Comments are more important than code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation. Queue 3(2):64–65
Ratol IK, Robillard MP (2017) Detecting fragile comments. In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE), IEEE pp 112–122
Reuveni D, Bourrillion K (2017) Code health: To comment or not to comment? https://testing.googleblog.com/2017/07/code-health-to-comment-or-not-to-comment.html. Accessed 28-04-2022
Saha K (2023) When should you add comments to your code? https://www.kinkarsaha.com/when-should-you-add-comments-to-your-code/. Accessed 28-04-2022
Schults C (2017) Learn which types of comments should be avoided. https://carlosschults.net/en/types-of-comments-to-avoid/. Accessed 28-04-2022
Sen A (2023) Optimal survey length: How long survey lengths can affect data quality. https://medium.com/think-cult/optimal-survey-length-how-long-survey-lengths-can-affect-data-quality-f0d6398d25ee. Accessed 03-10-2022
Shinyama Y, Arahori Y, Gondow K (2018) Analyzing code comments to boost program comprehension. In: 2018 25th Asia-Pacific software engineering conference (APSEC), IEEE pp 325–334
Sorens M (2017) Fighting evil in your code: Comments on comments. https://www.red-gate.com/simple-talk/opinion/opinion-pieces/fighting-evil-code-comments-comments/. Accessed 28-04-2022
Sourour B (2017) Putting comments in code: the good, the bad, and the ugly. https://www.freecodecamp.org/news/code-comments-the-good-the-bad-and-the-ugly-be9cc65fbf83/. Accessed 28-04-2022
de Souza SCB, Anquetil N, de Oliveira KM (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, pp 68–75
Spertus E (2021) Best practices for writing code comments. https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/. Accessed 10-04-2022

Sridhara G (2016) Automatically detecting the up-to-date status of todo comments in java programs. In: Proceedings of the 9th India software engineering conference, pp 16–25
Steidl D, Hummel B, Juergens E (2013) Quality analysis of source code comments. In: 2013 21st International conference on program comprehension (ICPC), IEEE pp 83–92
Storey MA, Ryall J, Bull RI, Myers D, Singer J (2008) Todo or to bug. In: 2008 ACM/IEEE 30th International conference on software engineering, IEEE pp 251–260
Struyf E (2021) To write code comments or not, it should not be a question. https://techcommunity.microsoft.com/t5/microsoft-365-pnp-blog/to-write-code-comments-or-not-it-should-not-be-a-question/ba-p/2178622. Accessed 28-04-2022
Stulova N, Blasi A, Gorla A, Nierstrasz, O (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 pp 65–69
Tan L, Yuan D, Krishna G, Zhou Y (2007) /* icomment: Bugs or bad comments?*. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp 145–158
Tan L, Yuan D, Zhou Y (2007) Hotcomments: how to make program comments more useful? HotOS 7:49–54
Tan SH, Marinov D, Tan L, Leavens GT (2012) @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth international conference on software testing, verification and validation, IEEE pp 260–269
Tenny T (1988) Program readability: Procedures versus comments. IEEE Trans Software Eng 14(9):1271
Tozzi C (2020) 5 code commenting don'ts. https://www.itprotoday.com/development-techniques-and-management/5-code-commenting-donts. Accessed 28-04-2022
Trivedi D (2019) Don't comment your code, rewrite it! https://devdeejay.medium.com/dont-comment-your-code-rewrite-it-a145d655f87b. Accessed 28-04-2022
Troisi M (2023) Useless comments can ruin your code reviews. here's how to erase them. https://techbeacon.com/app-dev-testing/useless-comments-can-ruin-your-code-reviews-heres-how-erase-them. Accessed 28-04-2022
Vogel P (2013) No comment: Why commenting code is still a bad idea. https://visualstudiomagazine.com/articles/2013/07/26/why-commenting-code-is-still-bad.aspx/. Accessed 28-04-2022
Wang D, Guo Y, Dong W, Wang Z, Liu H, Li S (2019) Deep code-comment understanding and assessment. IEEE Access 7:174200–174209
Wang R, Wang T, Wang H (2019) Study of a code comment decision method based on structural features. In: 2019 International conference on intelligent computing, automation and systems (ICICAS), IEEE pp 570–574
of Washington U (2023) Commenting. https://courses.cs.washington.edu/courses/cse142/21su/quality/commenting/. Accessed 03-10-2022
Wen F, Nagy C, Bavota G, Lanza M (2019) A large-scale empirical study on code-comment inconsistencies. In: 2019 IEEE/ACM 27th International conference on program comprehension (ICPC), IEEE pp 53–64
Wong E, Yang J, Tan L (2013) Autocomment: Mining question and answer sites for automatic comment generation. In: 2013 28th IEEE/ACM International conference on automated software engineering (ASE), IEEE pp 562–567
Wong E, Zhang L, Wang S, Liu T, Tan L (2015) Dase: Document-assisted symbolic execution for improving automated software testing. In: 2015 IEEE/ACM 37th IEEE International conference on software engineering, vol 1, IEEE pp 620–631
Woodfield SN, Dunsmore HE, Shen VY (1981) The effect of modularization and comments on program comprehension. In: Proceedings of the 5th International conference on software engineering, pp 215–223
Woost A (2011) Don't comment your code! https://apdevblog.com/comments-in-code/. Accessed 28-04-2022
Zhai, J., Huang J, Ma S, Zhang X, Tan L, Zhao J, Qin F (2016) Automatic model generation from documentation for java api functions. In: 2016 IEEE/ACM 38th International conference on software engineering (ICSE), IEEE pp 380–391
Zhai J, Xu X, Shi Y, Tao G, Pan M, Ma S, Xu L, Zhang W, Tan L, Zhang X (2020) Cpc: Automatically classifying and propagating natural language comments via program analysis. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering, pp 1359–1371
Zhang J, Xu L, Li Y (2018) Classifying python code comments based on supervised learning. In: International conference on Web information systems and applications, Springer pp 39–47
Zhané J (2023) Writing comments in python (guide). https://realpython.com/python-comments-guide/#python-commenting-best-practices. Accessed 28-04-2022
Zhong H, Zhang L, Xie T, Mei H (2009) Inferring resource specifications from natural language api documentation. In: 2009 IEEE/ACM International conference on automated software engineering, IEEE pp 307–318