The Secret Life of Commented-Out Source Code

Tri Minh Triet Pham, Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada
{p_triet,jinqiuy}@encs.concordia.ca

ABSTRACT

Source code commenting is a common practice to improve code comprehension in software development. While comments often consist of descriptive natural language, surprisingly, there exists a non-trivial portion of comments that are actually code statements, i.e., commented-out code (*CO code*), even in well-maintained software systems. Commented-out code practice is rarely studied and often excluded in prior studies on comments due to its irrelevance to natural language. When being openly discussed, CO practice is generally considered bad practice. However, there is no prior work to assess the nature of CO code practice: prevalence, evolution, motivation, and necessity of utilizing CO code practice.

In this paper, we perform the first study to understand CO code practice. Inspired by prior work in comment analysis, we develop automated solutions to identify CO code and track its evolution in development history. Through analyzing six open-source projects of different sizes and in diverse domains, we find that CO code practice is non-trivial in software development, especially in the early phase of development history, e.g., up to 20% of the commits involve CO code practice. We conclude common evolution patterns of CO code and find that developers may uncomment and comment code more frequently than expected, e.g., 10% of the CO code practices have been uncommented at least once. Through a manual analysis, we identify the common reasons that developers adopt CO code practices and reveal maintenance challenges associated with CO code practices.

KEYWORDS

commented-out code, comment analysis, comment/code evolution

ACM Reference Format:
Tri Minh Triet Pham, Jinqiu Yang. 2020. The Secret Life of Commented-Out Source Code. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389259

1 INTRODUCTION

Source code comments play an important role in software development and maintenance, e.g., helping developers document and

```
1 protected void importClass(String importPackage,
2     String name, String as) {
3 ...
4 -   // module.addImport( as, name );
5 +   output.addImport( as, name );
6 ...
7 }
```

Figure 1: An example of CO code in Apache Groovy where the variable *module* is no longer relevant, but the newly-added code logic is similar to the CO code.

```
@AfterClass
public static void cleanup() {
-  //clearProperty("atmosphere.disabled");
+  clearProperty("atmosphere.disabled");
}
```

Figure 2: An example of uncommenting CO code to fix bugs in test cases. The flaky tests are reported in CXF-8086 and CXF-8087.

understand the code logics and ease maintenance efforts [2, 8, 12, 22–24]. Comments often constitute of natural language descriptions to complement the expression ability of code. Since comments are not executed, developers may utilize this feature of comment to temporarily store unreachable code. Code is embedded as part of comments and is temporarily harbored in code repositories for potential later use. We call the code lines that are placed in comments as *commented-out code*, i.e., CO code for short.

CO code practice is controversial in the public eye. For example, we find many heated discussions on the usage of CO code on StackOverflow using the keyword "commented out code". There are many possible reasons to explain why CO code is considered controversial. First, CO code is not actively maintained, so even if the contained code logic is later reused, the exact code is not up-to-date with the latest version, e.g., the relevant live code may have undergone refactoring. Figure 1 shows an example of CO code from *Apache Groovy*. The commit adds a new statement (line 5), and its code logic is similar to the CO code line in line 4. However, the CO code in line 4 is not actively maintained after being commented out, i.e., *module* in line 4 is no longer available and is renamed to *output* in live code. Second, checking in incomplete code is considered as an invalid archive strategy. Instead, modern version control systems or tooling support on feature toggling should be utilized to achieve the same goal.

While there exist valid concerns on CO code practice, there are cases where CO code practice may help developers *effectively* handle disruptions during the development. However, such effectiveness comes with a cost: Developers may overlook the instability of CO code and forget to stabilize the temporary solutions that use

CO code. For example, when fixing bugs, developers may comment out and modify buggy lines to produce temporary patches. Since developers are not sure about the quality and stability of the fix, they may prefer to leave the original code logic in comments. Figure 2 shows an example of utilizing CO code in *Apache CXF*. When updated to work with JDK 11, line 3 was commented out as part of the fix for *CXF-7741* that can quickly address the *symptom* of unsuccessful build. Note that this temporary fix does not fix the root cause. Moreover, this temporary fix introduces flaky tests. The CO code was not uncommented until years later after several flaky tests were reported (i.e., CXF-8086 and CXF-8087). The CO code in this example is useful to help developers quickly decide how to fix the reported flaky tests.

While many prior studies [7, 16, 20] have analyzed comments comprehensively, there has been no work that carefully examines the practice of CO code. Most of the prior work focuses on the natural language descriptions in comments. Since CO code is in the format of code, and therefore is considered not interesting in prior studies. However, CO code practice, as a controversial approach that developers may utilize during software development, should be studied to understand the current practice and reveal potential maintenance challenges of CO code practice.

In this work, we perform the first study to understand the practice of utilizing CO code in software development. We developed lightweight approaches that specialize in analyzing CO code. The developed tool detects instances of CO code in a stable software version and tracks the history of CO code instance in the entire development history, i.e., when the CO code is first introduced (*comment-out*), when the CO code is brought back as live code (*uncomment*) and when the CO code is completely deleted from the code repository. Our evaluation shows that the developed tool achieves satisfying accuracy so that we can apply the tools to perform a systematic study of CO code practice.

In total, we study CO code practices in six open-source projects of different size (LOC) and from diverse domains: *VLC Android* and *NewPipe* are two mobile applications from F-Droid dataset [3]; *Groovy*, and *CXF* are desktop applications from Apache foundation [4]; *JUNG* is a popular network/graph framework that is used in a previous comment classification work [20]; and SWT is a graphical interface toolkit for Eclipse [5].

Our study focuses on understanding CO code practice from the following aspects: how often developers utilize CO code, i.e., the prevalence and evolution, how likely CO code lines co-transit, and why developers utilize CO code, i.e., the motivation of introducing CO code and uncomment CO code. In particular, we answer the following four research questions (RQs).

- **RQ1: How prevalent is CO code practice?** Given the reputation of CO code, it is interesting to learn how prevalent CO code is in well-maintained open source repositories. We find that CO code is not very prevalent in the latest version, however CO code is actively used (i.e., being introduced, moved and uncommented) in the development history.
- **RQ2: What are the evolution patterns of CO code?** Since we know CO code is often modified, we are interested in learning the lifecycle of CO code, and determine which evolution patterns they follow. We find that most of the CO code is introduced and removed. However, a non-trivial portion of them is uncommented to reuse the code logics of CO code.
- **RQ3: How often do CO code lines co-transit?** After we learned about the lifecycle patterns of the CO code, we continued to understand whether the CO code instances would co-transit with each other over their lifecycles. We find that CO code instances often do not co-transit.
- **RQ4: What are the purposes of utilizing CO code practice?** As we know that CO code is frequently used, we continue to unveil developers' motivations of utilizing CO code practice. We find that majority of them are motivated by two goals which are to introduce new code and to remove code. The remaining purposes include but not limited to code reversion, bug fixing, or temporarily code disabling for experiments.

Paper Organization. The rest of the paper is organized as follows. Section 2 describes our approach to detect and track the history of CO code and an evaluation of the developed approach. Section 3 describes the studied Java open-source projects and the motivation, method, and results of each RQ. Section 4 summarizes the threats to validity. Section 5 lists the related work on comment analysis. Finally, Section 6 concludes this study and describes future work.

2 APPROACH

In this section, we describe our automated approaches to extract the instances of CO code and track the history of such instances in code repositories. For each of the approaches, we evaluated the accuracy of the approach on a statistically significant data set.

2.1 Definition of CO Code: Our Studied Scope

For this paper, we define commented-out source code as an independent, syntactically correct code statements that were commented out. The following code elements are excluded from this study.

- Inline code that is not independent e.g., extra arguments in function calls that are commented out, and part of a statement that is commented out.
- Partial code that is not syntactically correct e.g., missing a semi-colon.

2.2 Identifying Commented-Out Code

Comment Extraction. First, our approach needs to identify all the Java files in an open-source project. Second, we utilize the Python module *Comment Parser* [1] to extract all the comment lines from the Java files identified in the previous step. *Comment Parser* extracts comments for many languages such as C/C++, Java, Python. For Java files, *Comment Parser* extracts both the single-line comments (i.e., marked by // and /**/) and multi-line comments (i.e., marked by /**/).

Pre-processing the Extracted Comments. After extracting all comment lines from Java files, our tool first merges consecutive comment lines into comment blocks. It is necessary to merge comment lines into blocks so that our tool does not miss CO code cases

[1]Comment Parser: a Python module to extract comments. https://github.com/jeanralphaviles/comment_parser

where the embedded CO code in comments spans multiple comment lines. The code snippet below shows one method invocation that spans multiple lines.

```
//An example of CO code instance that spans multiple comment lines, note
       that comment symbols // are omitted for clear illustration.
this.runInInterceptorAndValidate(
   "signed_issuer_serial_token.xml",
    PROTECT_TOKENS,
    SIGNED);
```

Second, we filter out all the comments about license and Javadoc [15]. Such comments may contain code elements. However, such "CO code" instances are less interesting to study since they are more likely to be stable and unlikely used by developers to perform some temporary tasks, i.e., resulting comment and uncomment activities and indicating maintenance challenges. Hence, we excluded all the comments on license and Javadoc from our analysis.

Extracting CO Code from Comments. We adopt a lightweight regex-based approach to detect CO code, which is different from previous work on comment analysis [16, 20]. We decided to pursue a slightly different approach because of a few concerns. First, previous studies that use machine learning to categorize comments, e.g., Steidl et al. [20], Pascarella and Bacchelli [16], focus on all comment categories. CO code is one of the categories and a less important one compared to other more significant categories. As a result, although the performance of all categories is sufficient, the accuracy of identifying CO code is less satisfying, and such inaccuracy will impact our study results significantly. Second, prior work classifies the comments at the line level and it is unclear whether the classification is still accurate for block level. Since CO code may span multiple comment lines, adopting machine learning-based approaches from prior work may not work well for our goal.

Our tool uses regular expressions [2] (which are inspired by the regular expressions used in [1, 20] but modified and expanded upon to directly match source code in comments' text) to identify CO code from comment blocks. For each comment block, if it contains *any* of the following escaped patterns, this comment block is considered to contain CO code by our tool.

(1) This regular expression is used to detect method calls: `[a-zA-Z0-9]+\s*\(.*?\)\s*?(\;|\{)`
(2) Declarations: `(public|private|void|protected|new)\s+.+?(\;|\{)`
(3) Code that can be followed by a block of code: `(if|while|for|catch)\s*\(.*` and `(else|do|try|finally|switch)\s*\{.*`
(4) Common statements: a return statement (`return\*\.*?\;`), or an assignment statement (`=\.+?\;`).

Evaluating the performance of our CO extraction tool. To evaluate the accuracy of our CO code extraction tool, we need to obtain the ground truth: for each comment line, we need to know whether it is a CO code instance. However, due to the large number of comments in the studied projects, we took a statistically significant (95% ± 5%) sample of *comments* and manually determine the ground truth. For most of the comments, it is straightforward to determine whether the comment contains CO code. However, for a few cases, when a very short code element is embedded in a regular comment line, whether or not it is a CO code instance becomes less clear. For such cases, we read the surrounding code and comments to make a judgment. In total, we sampled 2053 comments from the six studied projects. We measure the performance of our CO code extraction tool using two metrics: precision and recall. Precision is the percentage of the CO code extracted by our tool that is a *true* CO code instance. Recall is the percentage of all the *true* CO code instances that are extracted by our tool. Table 1 shows the precision and recall of our CO code extraction approach on a statistically significant sample on *comments*. We achieve 100% in recall for all the studied projects. For most of the evaluated projects, the precision is satisfying, i.e., over 97%. For Apache Groovy, the precision in identifying CO code is 84%. We examined the false positives in detail and discovered that in Groovy, the developers use many syntactically correct code snippets in the comments for documentation purposes. Groovy provides support to simplify Java programs. Hence developers use examples of Java code snippets to show what are the functionalities of some Groovy source code files. Since such uses of code in Groovy are mainly for documentation, similar to license and Javadoc, we mark them as false CO code. The accuracy of our CO code extraction tool (97%) is comparable to or even better than prior work that completes a similar task (89% [20] and 91% [16]).

[2]Regular Expression syntax based on Python re library: https://docs.python.org/3/library/re.html

Table 1: The precision and recall of our CO code extraction tool.

Name	#Cmts	#CO Code (*Ground Truth*)	Precision	Recall	F_1
VLC Android	245	12	100%	100%	100%
NewPipe	330	4	100%	100%	100%
JUNG	342	85	100%	100%	100%
Apache Groovy	375	43	84%	100%	91%
Apache CXF	380	35	97%	100%	98%
Eclipse SWT	381	29	97%	100%	98%

2.3 Tracking the History of CO Code

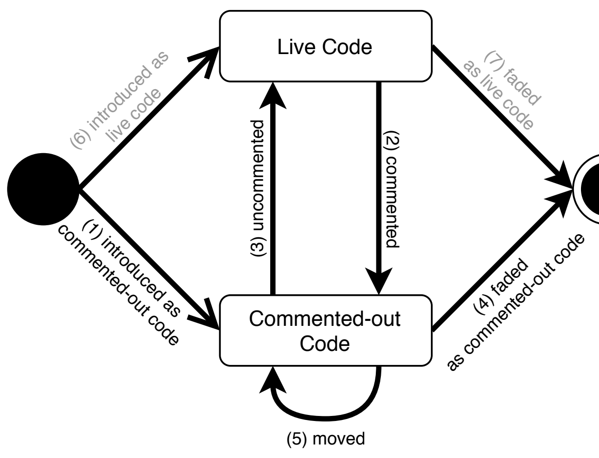
Live Code
Commented-out Code
(6) introduced as live code
(7) faded as live code
(3) uncommented
(2) commented
(1) introduced as commented-out code
(4) faded as commented-out code
(5) moved

Figure 3: Possible transitions of commented-out code

We developed an automated approach to track the history of CO code instances in code repositories and to assign a transition label to each of the CO-related changes. Note that a commit may contain many changes on CO code. The transition label is used to understand the evolution patterns of CO code practice.

Identifying CO Code Changes. We utilize *pydriller* [19] to iterate all the commits in the entire development history (i.e., from the first commit until the latest commit as shown in Table 3). In particular, our tool identifies all the commits that modify Java files. For each modified Java file in the identified commits, we use the tool described in Section 2.2 to extract CO code practices from the two versions: before and after the commit is applied.

Labeling the CO Code Changes. We assign each CO code transition a label. Such labels on CO code transitions include *introduce*, *uncomment*, *comment*, *move*, and *fade*. We describe the possible transitions between possible statuses of CO code instances in Figure 3. In addition to the transitions labels on CO code changes, Figure 3 includes two additional transition labels that are not directly on CO code instances, but on the code instances had at one point in the commit history been part of a CO code instance.

Our approach of assigning a transition label to a CO code change is based on matching the extracted CO code instances in the two versions of one commit. The matching would help us decide the exact transition label of each CO code change. First, we extract and transform the *code elements* in CO code instances to perform a better matching process. We summarize the transformations as follows: 1) We remove all the comment symbols, e.g., /*. 2) We remove all the empty spaces from the heading and tailing of the code element, e.g., "_", tabs, and newlines. 3) For the cases of nested comments (i.e., separated by (//) on the same comment line), we extract the code element of the nested comment (i.e., after the nested //), as a separate code element. Using 3), we can track each part of a nested CO code instance so that when a part is moved to a new location, our tool will correctly identify the transition label *move* instead of *introduce*. Note that we also apply the second transformation to regular *code* lines so that we can perform a more accurate matching between CO code elements and regular *code* lines.

After the previous extraction and transformation, for each version, we obtain two lists: *list_CO*, which contains all the CO code lines and *list_live*, which contains all the code lines that are not in comments. As a result, for each file in a commit, we obtain four lists of code. From the version before the commit, we obtain *list_CO_before* and *list_live_before* and from the version after the commit, we obtain *list_CO_after* and *list_live_after*. Then, our approach tries to match the two lists of CO code instances (i.e., *list_CO_before* and *list_CO_after*) one-by-one and label each of the CO code change with one the five transition labels as shown in Figure 3, i.e., *introduce, comment, move, uncomment, fade*. For each CO code element in *list_CO_before*,

(1) If it still exists in *list_CO_after*, the corresponding CO code transition is marked as *move*. The CO code instance is moved to other code locations in the same source-code file.
(2) If we cannot find the exact same code element in *list_CO_after*, then it indicates the removal of the CO code line. If the CO code line can be found in *list_live_after*, and the number of lines that are identical to the CO code line in *list_live_before* is equal to the number of lines that are identical to the CO code line in *list_live_after* + the number of live code lines identical to the CO code lines that are removed from *list_live_after*, the CO code change is marked as *uncomment*.
(3) Otherwise, if we cannot find the CO code in both *list_CO_after* and *list_live_after*, then it means that it is completely removed as CO code from the project source code (*fade from comment*).

Table 2: The precision of assigning transition labels to CO code changes.

Name	# CO code changes examined	Precision
VLC-Android	342	93%
NewPipe	310	92%
JUNG	350	95%
Apache Groovy	377	94%
Apache CXF	376	93%
Eclipse SWT	382	99%

We then repeat a similar process by iterating each code element in *list_CO_after*. If the code element can be found in *list_live_before*, the CO code change will be marked as *comment*. If the code element cannot be found neither in *list_live_before* nor in *list_CO_before*, the CO code change is marked as *introduced as commented-out code*.

Discussions. An alternative approach to process and label CO code changes per commit is to analyze the *diff* format of the commit. However, since the *diff* output is generated by existing diff algorithms, which may not consider the peculiarity of comments, the diff output on comments may yield inaccurate results. For example, if a few consecutive code statements are commented out using /*...*/, the diff output will only show that the first and the last lines are deleted, i.e., commented-out, while the lines in between remain as unchanged. Below is an example commit from *VLC-Android* for which the diff output fails to highlight that the lines between /*...*/ are deleted in the commit. Hence, we chose not to work on the diff output directly.

```
- String sql = "CREATE TABLE IF NOT EXISTS "+
+ /*String sql = "CREATE TABLE IF NOT EXISTS "
PLAYLIST_MEDIA_TABLE_NAME + " (" +
...
- db.execSQL(createPlaylistMediaTableQuery);
+ db.execSQL(createPlaylistMediaTableQuery);*/
```

Evaluating our approach that tracks the history of CO code. For the commits with assigned transition labels on CO code changes by our approach, we manually examine each transition label and decide the precision of our approach on a statistically significant sample (95% ± 5%). In total, we manually examined 2137 CO code changes and decide whether the assigned transition label is correct.

Table 2 shows the precision of the assigned transition labels in the studied projects: the percentage of CO code changes with correct transition labels assigned by our approach. The sampled transitions include various types of CO code, e.g., uncomment and comment. The precision of the assigned transition labels ranges from 92% to 99%, which shows that our approach achieves reasonable precision in tracking the history of CO code. The two main reasons are that some *faded as comment* CO code changes are mistakenly labeled as *uncommented* and some CO changes that *introduced as comment*

Table 3: Statistics on the studied projects. Note that all the statistics are calculated on Java files.

Name	Studied Version	LOC	# Cmts (License and Javadoc included)	# License	# Javadoc	**# CO Code** (# CO Code/# Cmts)	# CO code commits (# CO code commits/# all the commits)
VLC Android	0e94770	8K	2498	639	1183	32 (1.3%)	273 (4.4%)
NewPipe	4a76ba4	48K	4322	826	1187	54 (1.2%)	201 (4.0%)
JUNG	bf7e5b9	48K	14488	2617	8752	775 (5.3%)	70 (19.3%)
Apache Groovy	f105fc3	190K	92121	26991	49469	1491 (1.6%)	1008 (6.2%)
Apache CXF	b79021f	39M	189216	131792	22168	2231 (2.4%)	1017 (6.7%)
Eclipse SWT	871c71d	381K	197918	28747	129246	3015 (1.5%)	2017 (7.2%)

are mistakenly labeled as *commented*. Although we tried to remove some false positives in this type by handling duplicate code lines, we are not able to completely remove false positives.

3 STUDYING CO CODE PRACTICE

In this section, we answer four RQs based on the study on six diverse open-source Java projects. For each of the RQs, we present the motivation, method, and results.

Studied Projects. We analyze six open-source software systems in Java. The selected projects are of different sizes and in diverse domains. Specifically, we selected the six projects according to the following criteria. First, we started with the list of projects that are frequently analyzed by prior work on comment analysis [16, 20]. Such projects are well analyzed by prior work on comment and will allow us to compare our approach of CO code extraction with prior work for performance. Second, we selected the projects that are well-maintained and have a long development history. Last, we diversified the studied projects by including the ones on different platforms, i.e., PC/workstation OS and mobile OS. Table 3 summarizes the detailed statistics of the studied projects. The size of the analyzed projects ranges from eight thousand to 39 million lines of code (LOC). We analyzed the latest versions when we started this study. VLC Android and NewPipe are Android projects from f-Droid [3]. SWT is from Eclipse [5]. The remaining projects are from Apache Foundation [4].

RQ1: How prevalent is CO code practice?

Motivation. CO code practice is usually perceived as a controversial engineering practice [6, 9, 10, 14]. Whether or not there exists CO code practice in well-maintained and mature open-source projects is unknown. It is possible that CO code practice is more prevalent in certain development phases compared to a recent and stable software version. In this RQ, we analyze the prevalence of CO code from two aspects: in a stable version and from the entire development history (i.e., at commit level).

Method. We assess the prevalence of CO code practice from two aspects. First, we applied our CO code extraction tool (described in Section 2.2) to identify CO code instances in the latest version (when we started the study). The evaluation (in Section 2.2) of our CO code extraction tool shows the approach achieves acceptable precision and recall (i.e., 96.3% and 100% respectively), which are comparable and even better than previous work. Second, we studied the prevalence of CO code practice in the entire development history of all the studied projects. In particular, we applied the tool described in Section 2.3 to the development history of each project. The developed tool iterates all the commits and identifies CO code changes in each commit. We are interested to know 1) how many commits involve at least one CO code change and 2) for different development periods, whether CO code practice plays a role at different levels of significance. To study 2), we divide all the commits that modify Java files from the development history of each studied project into intervals. For all the projects except for *JUNG* and *NewPipe*, each interval contains 100 commits. For *JUNG* and *NewPipe*, each interval consists of 50 commits due to the shorter development period as well as the fewer number of commits in the history. For example, there are totally 1,223 commits in *NewPipe*, which results 25 intervals in total. Differently, *Eclipse SWT* has a much longer development history with 23,135 commits, so the resulting plot has 232 intervals, each consisting of 100 commits. We summarized the activities (i.e., accumulative *LOC* of CO code) of all the CO code commits in each interval.

Results. Table 3 summarizes the total number of comment lines, the numbers of comment lines in license and Javadoc, and the total number of CO code lines for the studied projects. The majority of the comments are about license and Javadoc. Only a small portion of all the comments are code comments that explain code logics using natural language. Table 3 also shows the percentage of CO code lines among all the comments, ranging from 1.2% to 5.3%. After excluding license and Javadoc, the percentage of CO code lines in the studied projects ranges from 2.3% to 9.5%. The results shows that CO code instances are not prevalent in the recent versions.

Despite the small portion of CO code lines in the recent stable versions, we find that there exists a non-trivial portion of commits that involve at least one CO code change. Table 3 shows that ranging from 4.0% to 19.3% of all the commits modifying Java files in the studied projects include at least CO code change. Considering the small number of CO code compared to code lines (e.g., CO code in *VLC Android* takes only 4‰ of all the code lines), CO code changes are much more frequent.

In addition, we also analyze the CO code activities in different development periods in the entire development history. Figure 4 shows that how many code lines are being actively developed/maintained in the equally divided ten time intervals. In particular, we quantify the development/maintenance activities using two metrics: 1) how many CO code lines are being added (the white bars in Figure 4), i.e., by either the transitions *comment* or *introduce*; and 2) how many CO code lines are no longer commented out (the gray bars in Figure 4), i.e., by either the transitions *uncomment* or *fade*. Our study shows that for the studied projects, the activities on CO code are not equally distributed across the development history.

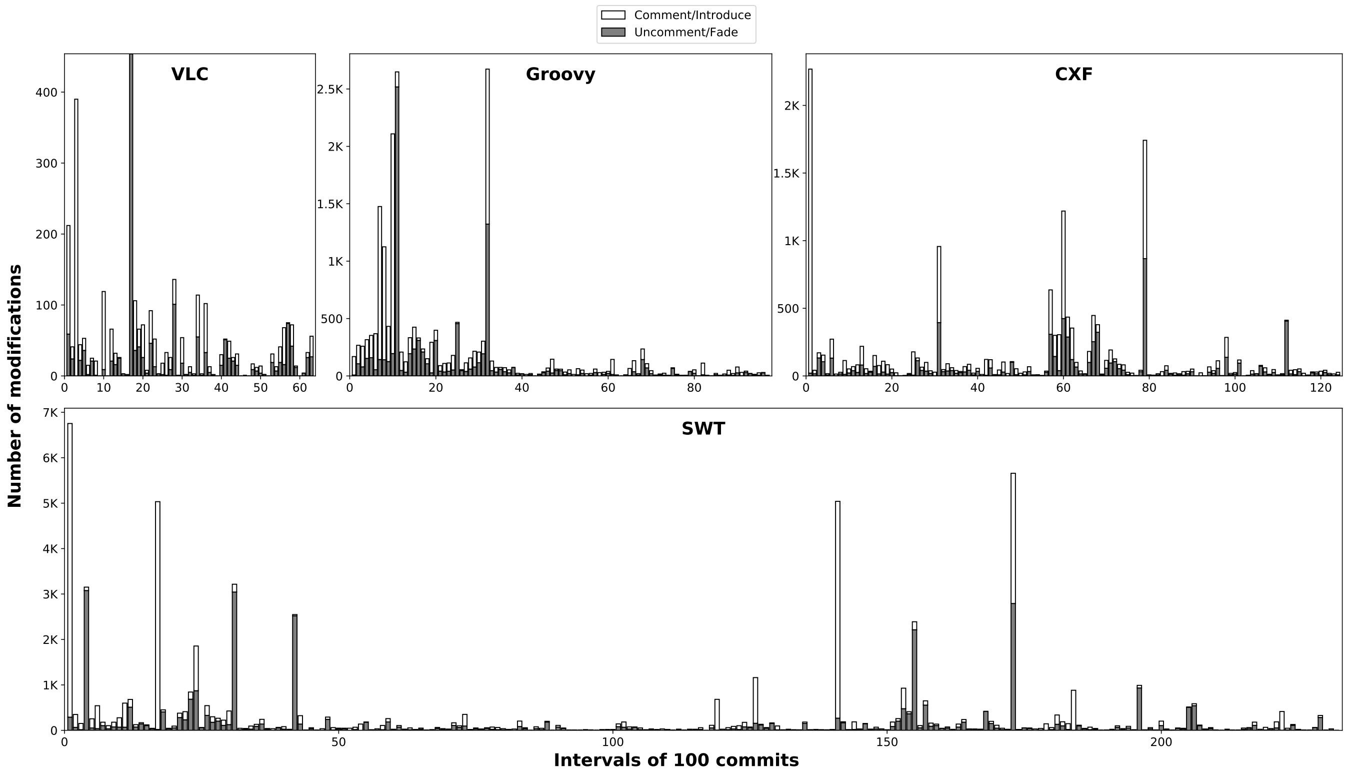

Figure 4: Accumulative modified LOC of CO code change commits in the entire development history for the projects where each interval contains 100 commits.

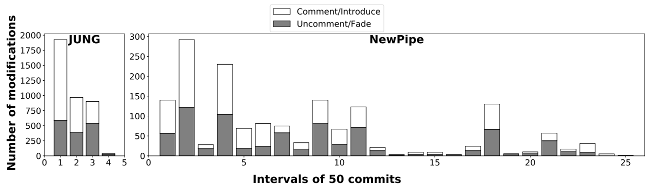

Figure 5: Accumulative modified LOC of CO code change commits in the entire development history for the two projects where each interval contains 50 commits.

For *JUNG*, *NewPipe*, *Groovy*, and *Eclipse SWT*, there exist certain development periods in which CO code practice is heavily used, while for most of the development periods, the CO code practice is not so commonly used. Differently, in *VLC Android* and *Apache CXF*, we find that CO code practice is steadily used in the entire development history.

Our study shows that despite the prevalence of CO code in a recent version is low, the prevalence of CO code in development history is non-trivial. We also find that in certain development periods, CO code is much more actively adopted than other development periods. This indicates the common use of CO code by developers for short-lived tasks.

RQ2: What are the evolution patterns of CO code?

Motivation. From the RQ1 of this study, we notice that there exists a non-trivial portion of commits that modify CO code. It means the transitions on CO code happen frequently, e.g., a code element is *commented*, followed by being *uncommented*. Developers frequently change the status of CO code as a temporary solution to many development tasks. However, simply examining the statistics from RQ1 does not provide detailed evolution patterns. Evolution patterns will provide a comprehensive understanding of the lifecycle of CO code instances. For example, how often developers add code *directly* as CO code for an on-going feature development? This motivates us to investigate the lifecycle of CO code. In particular, how one CO code instance evolves from the beginning to the end, i.e., removed completely from a code repository?

Method. We use the developed approach (described in Section 2.3) to track the history of each CO code instance. Our implemented approach assigns a transition label to each CO code change. Based on different combinations of transition labels, we observe four comprehensive evolution patterns that can cover all possible lifecycles. All four patterns start with the CO code is introduced either through 1) commenting out live code or 2) being introduced as CO code.

- **Pattern 1 (P1)**: <introduced as CO code | commented> → <moved>* (*optional*) → <faded as CO code>

A CO code instance is introduced. Then, it might be moved multiple times (optional), and eventually is completely deleted. The CO code instance has never been uncommented to become part of the live code.

- **Pattern 2 (P2)**: <introduced as CO code | commented> → <moved>* (*optional*)
  P2 is different from P1 only for the last transition. An instance of P2 remains as CO code in the project, unlike P1, in which the CO code is eventually removed.
- **Pattern 3 (P3)**: <introduced as CO code | commented> → <moved>* (*optional*) → <uncommented>
  The CO code has been uncommented at least once. This code still exists in the source code, either as CO code or live code.
- **Pattern 4 (P4)**: <introduced as CO code | commented> → <moved>* (*optional*) → <uncommented> → <moved>* (*optional*) → <faded as CO code | faded as live code>
  The CO code has been uncommented at least once. Unlike P3, this code is eventually removed from the source code.

To identify the lifecycle pattern of one CO code, we first analyze whether this CO code has a transition label of *uncomment* in its history. If it has *uncomment*, we confirm whether the CO code is eventually removed from the repository (**P4**) or not (**P3**). If the CO code does not have a transition label of *uncomment*, for the case that is eventually removed, we identify the lifecycle as **P1**, if not removed, the lifecycle pattern would be **P2**.

Results. Table 4 shows the details of CO code evolution patterns in the studied projects. Most of the CO code instances will never become live code again, i.e., P1 and P2 take the majority of the cases. Most of Pattern 1 cases are caused by commenting out code before deletion, usually as part of a large-scale refactoring or implementation change, e.g., a to-be-implemented feature is significantly modified. Pattern 2 means that the CO code instances remain in the repositories as of the newest version. They may become live code in the future. The P2 cases are typically caused by incomplete features that developers are still actively working on, e.g., the relevant test cases are commented out to avoid meaningless failures.

P3 and P4 stand for the CO code instances that have been *uncommented* at least once. Combining the cases in P3 and P4 takes a non-trivial portion of all the CO instances, i.e., ranging from 2% to 16%. The results show that developers leverage CO code practice to temporarily harbor some code logics which will later be reenabled.In fact, the utilization of code logics in CO code may be more frequent than what the union of P3 and P4 indicates. Recall the example shown in Introduction (Figure 1), CO code line may become obsolete but its code logic can still be reused by developers. Such cases are not included in the union of P3 and P4, while in the union of P1 and P2. In short, the re-utilization of CO code may be underestimated if we only look at P3 and P4.

Our analysis of CO code lifecycle patterns reveals that there exists a non-trivial portion of CO code instances, i.e., up to 16%, would be uncommented and become code again. However, we also notice that majority of the CO code instances will either remain in comments or are eventually deleted from source-code repositories.

Table 4: The distribution of four lifecycle patterns of CO code instances

Name	# Pattern 1	# Pattern 2	# Pattern 3	# Pattern 4
VLC Android	957 (80.0%)	41 (3.4%)	36 (3.0%)	162 (13.6%)
NewPipe	438 (63.4%)	66 (9.6%)	59 (8.5%)	128 (18.5%)
JUNG	1014 (56.3%)	580 (32.2%)	55 (3.1%)	153 (8.5%)
Apache Groovy	5841 (72.3%)	1348 (16.7%)	577 (7.2%)	308 (3.8%)
Apache CXF	2105 (34.9%)	2044 (33.9%)	1033 (17.2%)	843 (14.0%)
Eclipse SWT	20294 (78.2%)	2457 (9.5%)	393 (1.5%)	2823 (10.9%)

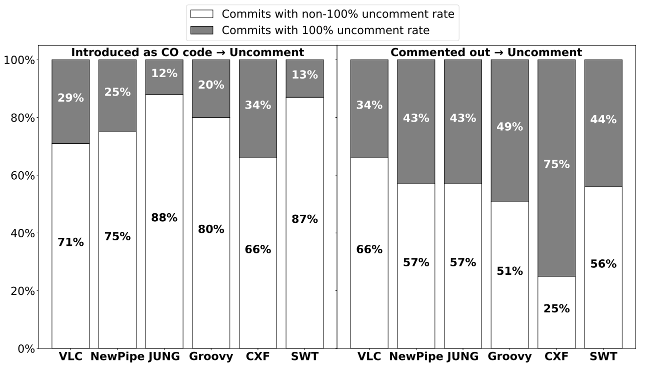

Figure 6: Plots to show the distribution of uncomment rate of the commits that uncomment at least one CO code line: the CO code lines that are introduced as CO code (the left side) and the CO code lines that are commented out (the right side).

RQ3: How often do CO code lines co-transit?

Motivation. RQ2 reveals that a non-trivial portion of CO code become live code (i.e., uncomment) again in software development. Since not all the CO code lines will be uncommented, it is likely that not all the co-transitioned CO code lines will co-transit again, i.e., becoming live code together by one follow-up commit. If CO code lines often do not co-transit, e.g., developers may gradually uncomment a few CO code lines in each follow-up commit, it indicates that uncommenting can be an error-prone practice and may need further tool support since developers need to precisely choose which lines (i.e., a subset of lines) to be live code again. Therefore we examine the co-transition status of CO code practices.

Method. We track co-transition status for two types of transitions: 1) *introduced as CO code* → *uncommented*; and 2) *commented out* → *uncommented.* For every commit that performs one of the transitions (i.e., *introduced as CO code* or *commented out*, we count the total number of CO code lines (*N*). We track the next steps (i.e., follow-up commits) of all the CO code lines introduced by the first commit. For each follow-up commit that uncomments any of the CO code lines, we count the total number of uncommented code lines (*n*). We name this percentage (*n/N*) as *uncomment rate.* For example, Commit-1 introduces five CO code lines. There are two possible scenarios: 1. A follow-up Commit-2 uncomment part of the five CO code lines (e.g., three lines), the uncomment rate is 60% (< 100%). 2. A follow-up Commit-2 uncomment all of the five

CO code lines, and its uncomment rate is 100%. We calculate the uncomment rate of all the follow-up commits and summarize the percentages of follow-up commits with 100% and non-100% for each studied project. If there are more follow-up commits with an uncomment rate of non-100%, the uncommenting practice may be more error-prone due to the ad-hoc and tedious nature.

Results. Figure 6 summarizes how often CO code lines co-transit, which is reflected by *uncomment rate* of the relevant follow-up commits. In particular Figure 6 highlights the percentages of commits with uncomment rate of 100% and non-100%. As we analyzed two types of transitions, we present the results separately, i.e., the CO code lines that are 1) *introduced as CO code* $\rightarrow$ *uncommented* (the left side) and 2) *commented out* $\rightarrow$ *uncommented* (the right side).

The co-transition results show that for both transition types, it is very often that CO code lines do not co-transition. Among the commits involving CO code that perform the first type of transition (i.e., *introduced as CO code*), for only 12% to 34% of them, all the CO code lines are sequentially *uncommented* in the same commit (100% uncomment rate). Similarly, except for CXF (75% of the commits having 100% uncomment rate), only 34% to 49% of the commits have an uncomment rate of 100%.

A non-100% uncomment rate indicates that the involved CO code lines are more scattered in the evolution, i.e., it may take several fellow-up commits to uncomment all the CO code lines that are initially introduced in the same commit. If the evolution of CO code instances is more scattered, it would be more challenging to perform uncommenting activities since the ad-hoc and tedious process may be more error-prone. Our study shows that many of the CO code-related commits are scattered, e.g., 87% of the relevant commits in Eclipse SWT involving gradually uncommenting part of the CO code lines.

Moreover, by comparing the two types of transitions, we find that the first type of CO code origin, i.e., "introduced as CO code", generally result more scattered CO code evolution, compared to the origin of "commented out". One reason we notice is that the total number of CO code introduced directly may contain a larger number of CO code lines, e.g., typically for implementing new features. An example of such practice is that developers add a chunk of code as CO code to implement a new feature, which is not completely ready due to various reasons, hence developers may need to gradually uncomment the CO code lines.

We find that CO code lines that are introduced in the same commit often do not co-transit together, i.e., scatteredness in the CO code evolution. It is common that for one commit that introduces many CO code lines, there are several follow-up commits to gradually uncomment part of the CO code lines. Such scattered CO code evolution may pose extra challenges since selecting which code lines to uncomment remains as an ad-hoc and tedious process.

RQ4: What are the purposes of utilizing CO code practice?

Motivation. Utilizing CO code practice has both advantages and disadvantages. Despite frequent usage, there are no empirical evidence on what exactly CO code practices are used for. Understanding developers' common needs of adopting CO code practice will provide empirical findings on how to improve the current status of CO code practices: Some CO code practices can be refactored out while some indicate maintenance challenges and developers may need more tool support to properly manage the evolution of CO code. For instance, if some CO code practices are used to implement feature toggling [18], they can be refactored by using feature toggling libraries, such as *Unleash* [3]. Another example usage of CO code is that developers may use CO code to annotate self-documented technical debt [17], then such CO code use may become outdated and cause extra burden, i.e., requiring non-trivial modifications once brought back as code. Obtaining a comprehensive understanding of the purpose of utilizing CO code is needed to inspire future research to provide better tooling support so that CO code practices will become less ad hoc.

Method. To understand the motivations behind the frequent use of CO code, we analyzed and categorized a statistically significant sample set of CO code transitions. In particular, we are interested in examining two types of transitions: commented out and uncomment. These two transitions play important roles in the lifecycle of CO code and obtaining a comprehensive understanding of the important transitions reveals the motivations of adopting CO code.

Table 5: The most common purposes to comment code and uncomment CO code. The rows are ranked according to how frequent each purpose is. Type annotates whether the CO code is test or code. Transition represents either commenting out (C) or uncommenting (U). CUT is short for code under test.

Type	Trans.	Purpose
Test	C	Temporarily disabled because of buggy CUT
Test	C	Temporarily disabled because the CUT is undergoing major modifications
Code/Test	C	Removed because the code and its tests are no longer in use
Code	C	Code used for debugging is disabled after fix bugs
Code	C	Removed because the design is changed
Code	C	Replaced by refactored code
Code	C	Removed in the process of adoption, migration, or removal of an external library
Code	C	Removed because the code contains bugs
Code	C	Removed because data structure/type is changed
Code	C	Removed because code is redundant
Code	C	Replaced by a new/improved functionality
Code	C	Temporarily removed until modifications are made to work with the new functionality
Code	U	Re-enabled because the buggy dependent code is fixed
Code	U	Enabled because dependencies are satisfied
Code	U	Reverted to a previous version because the new version is buggy, the design is changed, or code was commented out by mistakes
Test	U	Enabled/Re-enabled after bug in CUT is fixed
Test	U	Enabled/Re-enabled after the CUT is implemented or modified

[3]Unleash: an open source feature flag and toggle service. https://unleash.github.io/

For each project, we took a statistically significant (95%±5%) sample without replacement of all the transition instances. In total, we manually examined 342 *comment* transitions and 161 *uncomment* transitions. For each of the sampled CO code change, we analyzed the neighboring comments and code, the commit message and the corresponding bug report to determine why the CO change was adopted by developers: what is the reason that developers comment out code or introduce code as CO code; what is the motivation that developers decide to uncomment CO code. For a few cases that we analyzed, there is a lack of information for us to understand the motivation of introducing CO code, e.g., the commit message is empty or vague, or there are no corresponding bug reports. For such cases, we further analyzed the next transitions of the CO code instance and used them to reverse-engineer the motivation of the sampled transition.

Results. Our manual study (i.e., on 342 *comment* transitions and 161 *uncomment* transitions) reveals a list of motivations behind the transitions on CO code. When CO code is *introduced as CO code* (i.e., one type of comment transition), the most common motivation is to introduce new functionalities, however developers temporarily comment out such code since the dependent functionalities are not complete as a result of work in progress, pending refactoring or design changes. Therefore, developers harbour the code that implements the new functionality in comment section as CO code. The practice of adopting CO code for compromising incomplete relevant functionalities is quite common in both production code and test code. An interesting exception is that, in *VLC Android* we find a large number of CO code instances introduced due to importing the source code of an external library into the VLC codebase. Some other common usages include copying code templates from documentation that have CO code embedded, and for code debugging. The transition of *faded as CO code* is very straightforward, for which the primary purpose we find is to remove CO code and clean up codebase.

Table 5 summarizes the frequent motivations behind two transitions: comment and uncomment. We also annotate whether the CO code is test or non-test code.The most common motivation of commenting out code is to remove code for eventual deletion. However, there exist a number of cases where developers temporarily comment out the code to test new features or fix bugs where they are later uncommented. Corresponding, it is expected that such motivations of *comment out* have matching motivations of *uncomment*.

For the transition of *commented*, the most common motivations include marking the code for later permanent removal during non-trivial software maintenance efforts, such as refactoring, adding new functionalities, or design change. The less common yet prevalent motivations include to temporarily disable (i.e., comment out) the code to either experiment with new functionalities, cope with other code that is undergoing restructuring, or fix bugs. It is expected that the temporarily-disabled code would be brought back (i.e., uncomment) when the roadblocks are removed, e.g., bugs are fixed, or the dependent code is ready and complete.

For the transition of *uncomment*, there are two primary motivations. First, similar to the motivation of *introduced as CO code*, developers *uncomment* CO code to introduce the functionalities in CO code when the roadblocks are removed, i.e., the dependent code is ready. Second, we find that developers often *uncomment* CO code to revert the changes made by previous commits in which developers *comment out* source code. We also notice that developers may uncomment CO code both intentionally and unintentionally. Developers intentionally uncomment CO code to test new functionality, to toggle features, or to fix bugs. On the contrary, unintentional uncomment transitions may happen when CO code was initially commented out for later removal. However later, developers decided to discard the new implementation and revert to an earlier version, which leads to uncomment the CO code.

Our study shows that CO code practice is adapted for a wide variety of purposes. The primary motivation is to mark code for later removal. Meanwhile, developers commonly utilize CO code for aiding various development tasks quickly, such as bug fixing, code debugging and adding features.

4 THREATS TO VALIDITY

4.1 Internal Validity

CO code extraction. Per our definition of CO code, partial code (such as `node.getAttribute("w").equals("1")&&` or `, String id)` or `Set<Set<String>> clusterSet =`) is not considered. Capturing them would significantly decrease the precision of the extraction due to their incompleteness and diverse syntax which against our best interest of applying our method on new projects without having to manually re-calculate the precision and recall. Also, we do not use regular expressions to catch uncommon code statements. This can be a source of low recall. However, our data show that we can achieve almost the same precision and recall using just method matching and assignment matching, which suggest that having many extra regular expressions for matching is not necessary.

Transition detection. Transition detection recall is a source of concern. Given our methodology that ties CO code transition detection directly to *Git*'s ability to detect changes and our CO code detection recall (which is 100% as shown in Table 1), we are confident that we can retrieve all cases of transition.

Transition labeling. On the other hand, for transition labeling, the recall of each category is dependent on the precision of all other categories. In our case, we have cases of *fade* mistaken as *uncomment*, causing lower recall for *fade* and lower precision for *uncomment* compare to other categories. This is because of our method which compares the content of the line, transformed as described in the approach, so if we find 2 lines of identical CO code in the same file, even if they are on different lines, we still consider them the same because of the lack of context. Also, we cannot identify cases i.e. `ml.banFolder(x)` is refactored to `mediaLibrary.banFolder(x)` or `int Backward = 2;` is changed to `int Backward = 2;`.

4.2 External Validity

Our study is based on a set of popular open-source Java projects from GitHub and GitLab. While these projects are from different organizations, have different sizes, and are in different domains, our findings may not generalize to other projects, language or commercial software.

4.3 Construct Validity

Extraction validity. We reviewed our data and was surprised to discover that, had we only try to match method calls (1) and assignment statements (4), the precision and recall would not be much lower than the current results. This is interesting since 2 of the patterns are responsible for most of the findings. We try to craft patterns that can exhaustively discover all types of codes that present in comments. While this gives diminishing returns in increased running time and decreased precision, if we want to make sure to capture every single instance of CO code.

Labelling transitions as *move*. Our label *move* encapsulates 3 types of CO code movements in the file: other lines in the same CO code block is added or removed but the CO code line itself is not touched, the CO code line is shifted up or down due to changes to the other parts of the file, and CO code line is copied to somewhere else in the file. We decided that these 3 cases share the same label because: we do not distinguish the context of the CO code line; the key purpose of the label *move* is to show that the CO code line stays the same in the file, and *move* serves the purpose.

CO code transition purpose. While we tried to find as many purposes as possible, we may not cover all the possible purposes. These purposes were determined based on the combination of neighboring comments, commit messages, and issue trackers, all of which may not related to the CO code under examination. Some purposes can also be split or merged based on the experience of the investigator.

5 RELATED WORK

In this section, we summarize prior studies on comment and source code analysis, self-admitted technical debt and feature toggling.

Comments quality assessment and categorization. There have been many studies on comment analysis. Here we describe the most relevant studies that focuses on discussing the quality of comment and performing comment categorization. Haouari et al. [7] proposed a taxonomy of comments that includes categories in total. The categorization is based on four dimensions, i.e., object, type, style and quality. Steidl et al. [20] proposed a quality model and metrics to assess the quality of comments and surveyed to show the relevance of their metric to developers in practice. In particular, they proposed to use classification techniques to categorize comments automatically based on a mixture of location and purpose. Their classification approach achieves a precision of 89% and a recall of 95% for identifying CO code. Pascarella and Bacchelli [16] extended the work by Steidl et al. [20] by providing finer-grained comment categories. In addition, Pascarella and Bacchelli experimented different classification algorithms and showed promising results to classify comments into six major categories and 16 sub-categories.

These prior studies analyzed all types of comments and focus on comments in natural language. CO code, as one type of comments, is briefly discussed in prior studies but never comprehensively studied. For example, Haouari et al. [7] mentioned that outdated code is often commented out instead of removed. Pascarella and Bacchelli [16] mentioned that CO code instances envelop functional code to hide features, work in progress, features under test, etc. Steidl et al. [20] mentioned that CO code instances are temporarily commented out for debugging purposes or for potential later reuse. Our study complements prior studies on comment analysis by focusing on CO code. Also we developed a lightweight approach to identify CO code with very high precision and recall.

Source code detection from text. Tang et al. [21] used support-vector machine to extract non-text data from general email text and source code is one of the targets. Their approach uses 21 features to detect source code from each line of the email with a precision of 92.97%, and a recall of 72.17%. Bacchelli et al. [1] used regular expressions and pattern matching to extract source code from developer emails acquired from software system mailing list with a precision of 94% and recall of 85%. Similar to these prior studies, the goal of our developed approach is to identify code elements from text (i.e., comments). Differently, our task is simpler as CO code contains less non-code elements compared to emails. As a result, we can detect CO code with better precision and recall.

Self-admitted technical debt (SATD). SATD [11, 13, 17] is documented in comments. The SATD comments may contain CO code as shown in previous studies. Potdar and Shihab [17] examined the code that is intentionally introduced code as temporary workarounds. They extracted comments from the newest version of the projects under study and manually examined them to determine which comments are SATD. Liu et al. [13] and Huang et al. [11] provided tools to automate the SATD detection and management process using a text-mining based approach. However, not every CO code line is considered as SATD since SATD comments require to have keywords such as "TODO". In fact, in our dataset of CO code, we find that SATD makes up less than 6% of our dataset. Hence our study focus is different from previous studies on SATD with little overlap in the datasets.

6 CONCLUSION

Commented-out code is cited as a controversial coding practice. Despite the popularity of studies on comment, CO code is rarely studied and often excluded in previous studies. In this paper, we conducted the first study on CO code and focus on its prevalence, evolution, motivation, and potential maintenance challenges. We developed automated solutions to detect CO code instances and track their histories in code repositories. Moreover, we performed a manual analysis to understand the adoption of CO code practices and the challenges.

Our study on six open-source software systems reveals that despite the low prevalence of CO code in a recent version, CO code practice is actively used in certain development phases, i.e., up to 20% of the commits involving at least one CO code instance. Moreover by analyzing the evolution of CO code, we find that while CO code introduced in the same commit do not often co-evolve again, e.g., being uncommented together. Last, our manual analysis that there exist various maintenance needs so that developers need to utilize CO code practice. This paper presents insights on CO code practice that is previously overlooked, and sheds light on its prevalence and evolution in software development. Further studies are needed to provide tooling support to help developers better maintain and utilize CO code.

REFERENCES

[1] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. 2010. Extracting Source Code from E-Mails. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (ICPC '10)*. IEEE Computer Society, Washington, DC, USA, 24–33. https://doi.org/10.1109/ICPC.2010.47

[2] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information (SIGDOC '05)*. Association for Computing Machinery, New York, NY, USA, 68–75. https://doi.org/10.1145/1085313.1085331

[3] F-Droid. 2019. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/

[4] Apache Software Foundation. 2019. Apache Software Foundation. https://www.apache.org/

[5] Eclipse Foundation. 2019. Eclipse Foundation. https://www.eclipse.org/

[6] gaazkam (https://softwareengineering.stackexchange.com/users/212639/gaazkam). 2018. Why is it wrong to comment out code and then gradually remove it to keep track of what I've already done and what remains to be done? Software Engineering Stack Exchange. https://softwareengineering.stackexchange.com/q/377186 version: 2019-12-01.

[7] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How Good is Your Comment? A Study of Comments in Java Programs. *International Symposium on Empirical Software Engineering and Measurement*, 137–146. https://doi.org/10.1109/ESEM.2011.22

[8] Carl Hartzman and Charles Austin. 1993. Maintenance productivity: Observations based on an experience in a large system environment. 138–170. https://doi.org/10.1145/962304

[9] JoelFan (https://softwareengineering.stackexchange.com/users/213684/). 2013. good replacement for commenting out code? Software Engineering Stack Exchange. https://softwareengineering.stackexchange.com/q/213684 version: 2019-12-01.

[10] Alexis Dufrenoy (https://softwareengineering.stackexchange.com/users/8033/) and Thomas Owens (https://softwareengineering.stackexchange.com/users/4/). 2013. Can commented-out code be valuable documentation? Software Engineering Stack Exchange. https://softwareengineering.stackexchange.com/q/190096 version: 2019-12-01.

[11] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2017. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* (2017), To Appear.

[12] Zhen Ming Jiang and Ahmed E. Hassan. 2006. Examining the Evolution of Code Comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/1137983.1138030

[13] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In *Proceedings of the 40th International Conference on Software Engineering - Tool Demonstrations track (ICSE'18 Demos)*. IEEE.

[14] nikie (https://softwareengineering.stackexchange.com/users/14237/) and GBH (https://softwareengineering.stackexchange.com/users/38887/). 2011. Is commented out code really always bad? Software Engineering Stack Exchange. https://softwareengineering.stackexchange.com/q/190096 version: 2019-12-01.

[15] Oracle. 2019. Java Documentation: Javadoc. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html#CHDBEFIF

[16] Luca Pascarella and Alberto Bacchelli. 2017. Classifying Code Comments in Java Open-source Software Systems. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 227–237. https://doi.org/10.1109/MSR.2017.63

[17] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 91–100.

[18] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[19] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[20] D. Steidl, B. Hummel, and E. Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. 83–92. https://doi.org/10.1109/ICPC.2013.6613836

[21] Jie Tang, Hang Li, Yunbo Cao, and Zhaohui Tang. 2005. Email Data Cleaning. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. ACM, New York, NY, USA, 489–498. https://doi.org/10.1145/1081870.1081926

[22] Ted Tenny. 1985. Procedures and Comments vs. the Banker's Algorithm. *SIGCSE Bull.* 17, 3 (Sept. 1985), 44–53. https://doi.org/10.1145/382208.382523

[23] T. Tenny. 1988. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[24] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, 215–223.