RepliComment: Identifying Clones in Code Comments

Arianna Blasi[♠] · Nataliia Stulova[♣] · Alessandra Gorla[♡] · Oscar Nierstrasz[♣]

[♠] *USI Università della Svizzera italiana, Switzerland*
[♡] *IMDEA Software Institute, Spain*
[♣] *University of Bern, Switzerland*

Abstract

Code comments are the primary means to document implementation and facilitate program comprehension. Thus, their quality should be a primary concern to improve program maintenance. While much effort has been dedicated to detecting bad smells, such as clones in code, little work has focused on comments. In this paper we present our solution to detect clones in comments that developers should fix. RepliComment can automatically analyze Java projects and report instances of copy-and-paste errors in comments, and can point developers to which comments should be fixed. Moreover, it can report when clones are signs of poorly written comments. Developers should fix these instances too in order to improve the quality of the code documentation. Our evaluation of 10 well-known open source Java projects identified over 11K instances of comment clones, and over 1,300 of them are potentially critical. We improve on our own previous work, which could only find 36 issues in the same dataset. Our manual inspection of 412 issues reported by RepliComment reveals that it achieves a precision of 79% in reporting critical comment clones. The manual inspection of 200 additional comment clones that RepliComment filters out as being legitimate, could not evince any false negative.

Keywords: Code comments, Software quality, Clones, Bad smell

1. Introduction

It is standard practice for developers to document their projects by means of informal documentation in natural language. The Javadoc markup language, for instance, is the de-facto standard to document classes and methods in Java projects. Similar semi-structured languages are available for other programming languages. Given that many projects have code comments as the only documentation to ease program comprehension, their quality should be of primary concern to improve code maintenance. The quality of code comments is important also because there are many techniques that use comments to automate software engineering tasks, such as generating test cases and synthesizing

code [1, 2, 3, 4]. Without comments of high quality, the effectiveness of these techniques cannot be guaranteed.

Our research roadmap is to develop techniques to support developers in *identifying and fixing issues that affect the quality of comments*. As a starting point of our research, we have previously proposed RepliComment-V1,[1] a technique to identify and report *comment clones* [5]. Our main hypothesis is that clones in comments may be the result of bad practice, and just as clones in code, they should be identified and fixed.

Comment clones can highlight various issues: They may be instances of copy-and-paste errors, and therefore comments may not match their corresponding implementation. They may simply provide poor information, which may not be useful to understand the implementation. Our analysis shows that most of the time comment clones are signs of documentation that could be improved.

Corazza *et al.* conducted a manual assessment of the coherence between comments and implementation, and found instances of comment clones [6]. Similarly, Arnaoudova et al. [7] found some comment clone practices in their study about Linguistic Antipatterns in software. They report that 93% of interviewed developers considered documentation clones to be a poor or very poor practice. These studies show that the comment clone problem exists and is relevant for developers. Moreover, Aghajani *et al.* [8] suggest development of NLP-based techniques to identify cloned content, and suggest fixes in software documentation as a priority task within the software engineering community. It is finally worth noting that, in the code clone detection community, techniques that attempt to detect *code similarity* and *code clones* by means of API documentation are emerging [9, 10]. For such techniques cloned documentation would be highly deceptive, by falsely identifying functional similarities. We thus believe that approaches to automatically detect and fix problematic comment clones would provide an important service to the community.

We have previously presented RepliComment-V1 [5], a technique to automatically identify comment clones that may be symptoms of issues that developers want to fix. RepliComment-V1 suffers from several limitations. First and foremost it reports all found comment clones, except for few cases that trivial heuristics filter out. This causes many legitimate comment clones to be reported as needing to be fixed, while they are in fact just false positives. Moreover, RepliComment-V1 cannot pinpoint which comments are the original, correct ones and which ones are the clones to be fixed. In this paper we address these limitations. We present:

- new heuristics to filter out most false positives. Specifically, the new heuristics can accurately filter out 61,459 false positives, which amounts to +8% more cases that RepliComment successfully filters out compared to RepliComment-V1 [5].

[1] We will refer to the original version of RepliComment as RepliComment-V1, to distinguish it from the improved version we present in this paper.

- a novel implementation that looks not only for clones in method comments, but also in field comments.
- a parameterized analysis that looks for clones in various scopes of a Java project: intra-class, inter-class within the same class hierarchy, and inter-class across the entire project.
- a new component to classify comment clones by *severity*.
- a natural language processing technique to analyze the comment clones to pinpoint which comment block should be fixed.

We use the newly improved RepliComment to analyze the code base of 10 well-established Java projects. Our evaluation highlights that even solid and well-known projects contain comment clones. Specifically, we highlight over 11K comment clones, of which over 1,300 are critical and should be analyzed and fixed by developers with high priority to improve the quality of documentation. A qualitative analysis on a small sample of the results show that RepliComment achieves a precision of 79%, and the clones that RepliComment filters out are true false positives. Thus RepliComment can be trusted by developers to find and fix comment clone issues.

The remainder of this paper is structured as follows: Section 2 presents some real examples of comment clones, which may identify issues, or may be legitimate cases. Section 3 describes RepliComment and all its internal components to identify, filter and analyze comment clones. Section 4 presents the results of the evaluation of our extended technique, and a direct comparison with [5]. Section 5 discusses some related work, and section 6 concludes and discusses the future research direction of this work.

2. Comment Clones

Javadoc is a semi-structured language to document a class, its declared fields and its methods. Comments related to method declarations usually have a general description of their functionality, and then include specific tags describing each parameter, the return value and thrown exceptions, in case there are any.

Javadoc comments are often the only documentation available to understand the offered functionalities and the implementation details of a Java project. Therefore, their quality is important. Clones in comments, just as in code, may be a sign of poor documentation quality, and therefore should be identified and reported.

According to the state of the art taxonomy [11], code clones can be instances of Type I, *i.e.*, exact copy-and-paste clones, up to Type IV clones, *i.e.*, semantically equivalent code snippets. Comment clones can be classified according to the same taxonomy as follows:

Type I comment clone: The comment of a code element, *i.e.*, a method, a class or a field, is an exact copy of the comment of another code element except for whitespace and other minor formatting variations.

Type II comment clone: The comment of a code element is an exact copy of the comment of another code element except for identifier names.

Type III comment clone: The comment of a code element is an exact copy of the comment of another code element except for some paragraphs. For instance, the Javadoc comment of a method has the very same free text of another method, but the @param, @throws, or @return tag descriptions differ. Conversely, tag descriptions may be the same, and Javadoc comments may differ in the free text description of the method.

Type IV comment clone: The comment of a code element is lexically different, but semantically equivalent to the comment of another code element.

The fundamental difference with respect to code clones is that comment clones of any type are not necessarily an issue, and therefore they should not always be reported. Comment clones should be reported when they are the result of copy-and-paste errors, and the copied comment does not match the implementation of its corresponding code entity. Also, comment clones may exist because of the poor practice of developers of using generic, uninformative descriptions for multiple code elements in the same project. However, comment clones may also exist for justified reasons, for instance in case of method overloading, where the general description of the method is meant to be the same. Such instances of comment clones should not be reported.

RepliComment aims to find *problematic* Type I and Type III comment clones affecting methods and fields within the same class, across classes within the same hierarchy, or across classes within the whole project. RepliComment does not report Type II clones since comments differ in identifiers, and therefore likely document their corresponding piece of software correctly. We now present some real examples of comment clones that RepliComment can deal with.

A critical comment clone is that of a comment that is copied from a correctly documented method or field, and erroneously pasted to another code entity whose functionality differs completely. One example of this issue exists in the Google Guava project in release 19:[2]

In this example (see Sample 1), the Javadoc @return tag of method matchesNoneOf() is a clone of method matchesAllOf(), offered by the same class CharMatcher. It is easy to see that the return comment of the second method does not match the semantics of its name, while it does match the semantics of matchesAllOf(). This clone is clearly an example of a copy-and-paste error. It is conceivable that the developers first implemented method matchesAllOf(), and later implemented matchesNoneOf() starting from a copy of the first method. The two methods have a similar purpose, *i.e.*, to filter a collection of elements, however in the first case the filter returns all elements matching a given pattern, while in the second case it returns those that do *not* match the given pattern.

[2] http://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/CharMatcher.html

```
/**
 * @return true if this matcher matches every character in the
 * sequence, including when the sequence is empty.
 */
public boolean matchesAllOf(CharSequence sequence) { ... }
```

```
/**
 * @return true if this matcher matches every character in the
 * sequence, including when the sequence is empty.
 */
public boolean matchesNoneOf(CharSequence sequence) { ... }
```

***Sample 1**: Comment clone due to copy-and-paste error.*

Comment clones may also be examples of poor documentation that could be improved to offer a better understanding for developers. See the following example from a non-public class in the Apache Hadoop project release 2.6.5:

```
/**
 * @return true or false
 */
@InterfaceAudience.Public
@InterfaceStability.Evolving
public synchronized static boolean isLoginKeytabBased() throws IOException
{ ... }
```

```
/**
 * @return true or false
 */
public static boolean isLoginTicketBased() throws IOException { ... }
```

***Sample 2**: Comment clone of poor information.*

These two methods offered by class UserGroupInformation have exactly the same comment regarding the postcondition. It states that the methods return either true or false, which is correct. However, the documentation is uninformative, since any boolean method obviously returns either true or false. A more useful documentation should state what the boolean value represents, *e.g.*, whether it is a system component status, or the result of a conditional check. Such clones are symptoms of documentation that could be improved, and thus RepliComment aims to report them as well.

Not all comment clones are necessarily an issue to report to developers. They may occur for legitimate reasons, such as when two methods offer the same functionality. The following example comes from class SolrClient of the Apache solr library release 7.1.0:[3]

[3] `https://lucene.apache.org/solr/7_1_0//solr-solrj/org/apache/solr/client/solrj/SolrClient.html#deleteById-java.lang.String-java.lang.String-`

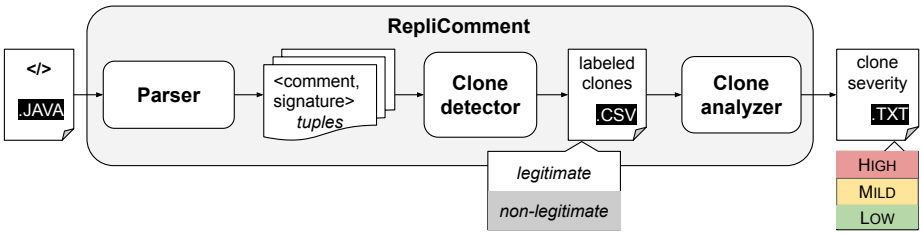

Figure 1: RepliComment components

```
/**
 * Deletes a single document by unique ID
 * @param collection the Solr collection to delete the document from
 * @param id the ID of the document to delete
 */
public UpdateResponse deleteById(String collection, String id) { ... }
```

```
/**
 * Deletes a single document by unique ID
 * @param id the ID of the document to delete
 */
public UpdateResponse deleteById(String id) { ... }
```

***Sample 3**: Legitimate comment clone due to method overloading.*

The clone in this case affects the free text in the Javadoc comments. Methods deleteById(), however, are an example of function overloading. Given that they have similar purposes, it is legitimate for their method descriptions to be identical. The difference between these two methods, which lies in their parameter lists, is properly documented through the custom @param tags.

3. RepliComment Components

Figure 1 shows at a high level the main components of RepliComment and its workflow. RepliComment analyzes an entire Java project, searching for code clones across various scopes. By default it looks for clones within the same class (*i.e.*, intra-class), however, upon changing the configuration, it can search for clones also across all Java classes, either within the same hierarchy (*i.e.*, intra-hierarchy) or across the whole project (*i.e.*, inter-class).

The *Parser* component (subsection 3.1) analyzes the input Java file and for each method declaration it produces a tuple of the method signature and corresponding Javadoc comment. Similarly, it produces a tuple for each class field declaration and its corresponding Javadoc comment. Next, the *Clone detector* component (subsection 3.2) takes the tuples produced by the *Parser* and uses several simple syntactic heuristics to filter out *legitimate* clones, marking the rest as being *non-legitimate*. Finally, the *Clone analyzer* component (subsection 3.3) investigates the *non-legitimate* comment clones. For each case the *Clone analyzer* computes the *severity* level of the clone and uses this to further categorize

the clone. Both HIGH and MILD severity levels indicate a *non-legitimate* comment clone, such as those resulting from copy-and-paste errors (as in Sample 1), or containing poor information (as in Sample 2), respectively. A LOW severity level can indicate a *legitimate* clone (as in Sample 3), or a false positive result of the analysis, *i.e.*, a case where comments are not actually clones of one another. We now describe each core component of RepliComment in more detail.

3.1. Parser

The *Parser* component of RepliComment takes as input a single Java file, identifies the list of declared methods and field, and stores all method signatures and field names. For each method and field it then identifies the corresponding Javadoc comment and parses it, extracting the following comment parts, if present:

free text: text in natural language, usually tag-free, typically present at the beginning of the block comment, providing a high-level description of the method or of the field.

@param tag: a method comment block describing a single specific parameter.

@return tag: a method comment block describing the return value of the method, when not void.

@throws tag: a method comment block describing possible exceptional behaviors. @exception tags are treated just like @throws tags.

The *Parser* is built using the JavaParser library.[4] It includes a pre-processing step that cleans each Javadoc paragraph. Specifically, it removes all whitespace as well as HTML code and any other semantically irrelevant Javadoc tags such as @see and @link. Such tags are not relevant for RepliComment, since they do not help in identifying which code identities the comment refers to, and are therefore discarded. The *Parser* outputs a list of tuples of field names and method signatures, and their respective pre-processed Javadoc comments, where each comment is reduced to a list of labeled comment parts described above.

3.2. Clone Detector

The *Clone detector* aims to identify likely comment clones and distinguish the *legitimate* and *non-legitimate* clones. It loops through all the method and field declarations identified by the *Parser* and looks for Type I clones of whole Javadoc comments. It then proceeds to detect type III clones, *i.e.*, clones of comment parts across different methods. Indeed, a single comment part may be cloned while the rest of the comment is not. In particular, a single comment part may be the free-text summary preceding the Javadoc tags, a @param tag, the @return tag, or a @throws or @exception tag.

[4] `https://github.com/javaparser/javaparser`

The *Clone detector* would thus flag a potential comment clone if two methods (or fields) use the same comment to describe the method (or field), either entirely or just in some parts. However, such a naive check is prone to false positives. Hence, this component uses several heuristics to filter out false positives and only flag real clone suspects. The *Clone detector* operates in two main steps:

1. It takes the tuples produced by the *Parser* (section 3.1), and compares each comment block with the same type of comment blocks of all the other methods withing the same file or across files, according to the desired scope. First, it compares whole Javadoc blocks to check whether there are *whole comment* clones documenting methods. This differs from RepliComment-V1 [5], which never looked for whole comment clones. Then, it proceeds with the comparison of single *comment parts*: it compares each `@param` tag comment with other `@param` comments and so on. For fields, the comment always consists of the free-text part only.

2. When the *Clone detector* finds that two or more clones of as Javadoc comment, it checks whether the clone might be *legitimate* or *non-legitimate*. RepliComment never considers whole Javadoc comment clones to be *legitimate*, and we explain why in section 3.3.

   RepliComment-V1 considered a cloned comment *part* to be potentially *legitimate* if it satisfied any of the following heuristics:

   - the clone is found in methods with the same (overloaded) names,
   - the comment describes the same exception type, or
   - the clones affect parameters that have the same name.

   RepliComment now additionally employs the following heuristics:

   - An exception comment must consist of at least 4 words and must not match a generic exception description pattern (recognized via a regex). We have observed that three words are insufficient to express the conditions under which an exception is thrown; furthermore certain generic patterns, such as *"@throws exception for any kind of error,"* are common.
   - The clone concerns `@return` tags of methods with the same, non-primitive return type. This is useful for filtering out APIs with methods that always update the class instance and return it, for which it is legitimate to have comments such as *"@return a reference to this."*
   - Constructors without parameters are allowed to have cloned comments, since they can have very generic comments, according to the official Oracle guide to writing good Javadoc documentation.[5]

[5] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#defaultconstructors`

- Fields with same name in different classes are allowed to have the same comment.

Finally, clones processed by the heuristics are stored in a csv report file as tuples with the following items:

- the fully qualified name of the class,
- the signature of the first method or field,
- the signature of the second method or field,
- the type of cloned Javadoc comment part (*i.e.*, whole, free-text, @param, @return or @throws),
- the cloned text, and
- a value indicating if the clone is considered *legitimate* or rather *non-legitimate* by the *Clone detector*.

The csv report is the input to the next component, which performs an analysis of the clone suspects to determine their severity level.

3.3. Clone Analyzer

The *Clone analyzer* [12] takes as input the csv file produced by the *Clone detector* and performs an analysis only on comment clones flagged as *non-legitimate*. Clones flagged as *legitimate* are ignored, trusting the judgment of the heuristics described in section 3.2. This way, the heuristics act as a filter on all the possible cases of comment clones that can be encountered in a Java project and may contain a high number of false positives. Since the *Clone analyzer* needs to perform a careful analysis on each suspect, the heuristic filter helps to significantly reduce the computational effort.

Clone analysis algorithm. We now describe how the *Clone analyzer* computes its analysis. We present its pseudo-code in algorithm 1, specifically referring to method comments since they are the most complex to deal with. When dealing with field names instead of method signatures, the reasoning about similarity thresholds is the same.

As we see in line 3 of algorithm 1, the *Clone analyzer* first checks whether the clone under analysis is a whole Javadoc comment clone. Such types of clones need special consideration. As the official Oracle guide to the Javadoc tool explicitly specifies, developers should *"write summary sentences that distinguish overloaded methods from each other"*.[6] Hence, when a *whole* Javadoc comment is cloned, RepliComment assumes there is some sort of issue no matter if the methods are overloaded or not. In other words, whole Javadoc comment clones

[6] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#doccommentcheckingtool`

Algorithm 1 Clone analyzer

```
1: /** Given a pair of method signatures and the cloned Javadoc comment, return the severity
   score of the clone as a warning */
2: function ANALYZE-COMMENT-CLONES(methodSignature1, methodSignature2, clonedJavadoc)
3:     if clonedJavadoc is of type WHOLE_JAVADOC_BLOCK then
4:         if IS-OVERLOADING(methodSignature1, methodSignature2) then
5:             REPORT(Please document parameter)
6:             WARN(MILD_SEVERITY) & EXIT
7:         else
8:             REPORT(Not overloading: fix these comments)
9:             WARN(HIGH_SEVERITY) & EXIT
10:    m1Sim = COMPUTE-SIMILARITY(methodSignature1, clonedJavadoc)
11:    m2Sim = COMPUTE-SIMILARITY(methodSignature2, clonedJavadoc)
12:    if  m1Sim < MIN-THRESHOLD and m2Sim < MIN-THRESHOLD then
13:        REPORT(Please fix poor info comment)
14:        WARN(MILD_SEVERITY)
15:    if m1Sim > 0.50 and m2Sim > 0.50 then
16:        REPORT(This looks like a false positive)
17:        WARN(LOW_SEVERITY)
18:    if | m1Sim - m2Sim| > DIFF-THRESHOLD then
19:        REPORT(Please fix method with lowest sim score)
20:        WARN(HIGH_SEVERITY)
21:    else
22:        REPORT(Fix these comments)
23:        WARN(HIGH_SEVERITY)
```

are never considered *legitimate* by the *Clone detector*, and are never labeled as LOW severity issue by the *Clone analyzer*. In case of overloading, the *Clone analyzer* flags such an issue as MILD severity, and RepliComment will report the problem suggesting the developer to correctly document the difference in the parameters. Otherwise, the *Clone analyzer* flags the issue as HIGH severity. We assume that there are major issues to fix if unrelated methods have the same comment.

In lines 10 and 11 of algorithm 1, the *Clone analyzer* computes the similarity scores between the cloned comment and each of the involved methods (we explain the details of this computation below). The similarity scores are used to determine whether the clone is a LOW, MILD or HIGH severity issue:

- Both methods can achieve a very low similarity score with respect to the cloned comment (line 12): the assumption is that the comment is so generic that it does not document well enough either of the methods. We set the MIN-THRESHOLD value to 0.25, based on empirical evidence that this value is the best balance to detect correct matches, while limiting false positives. This is a MILD severity issue, and the *Clone analyzer* requires the developer to add more detail to the comment for those methods.

- Both methods can achieve a very high similarity score with respect to the cloned comment (line 15): in this case the comment looks good enough for both. These cases were not filtered out by the heuristics of the *Clone detector* in section 3.2, but look like false positives nonetheless. Thus, they are reported to be LOW severity issues by the *Clone analyzer*.

- If none of the above cases hold, then first we consider the case where one

method achieves a significantly better similarity score than another. The method that achieves the highest similarity score is assumed to be the real owner of the comment, while the other is reported to be the victim of a mistaken copy-paste. We set the DIFF-THRESHOLD value to 0.1, once again due to empirical evidence. If both methods have very close similarity scores, both comments are reported as needing correction. Comment clones for which the owner is clearly distinguishable tend to be Type III clones, such as the one in Sample 2. Indistinguishable comments, instead, mostly belong to Type I clones, i.e. whole comment clones. Such comments are not overly generic, but at the same time, they are not informative enough to highlight the distinction between two different code elements. This case is reported as a HIGH severity issue, urging the developer to fix the wrongly-documented method(s).

We now expand on the description of how a similarity score between a method and its comment is computed.

Method-comment similarity computation. We take the full method signature and the part of the method comment marked by RepliComment as a likely clone and compute the similarity between them based on natural language cues present in each of them. Our underlying assumption here is that both the comment text and the identifiers in the signature (method name, parameter names, type identifiers *etc.*) are written in the same language. This allows us to rely on natural language processing (NLP) techniques to extract vocabularies of each entity, and use the similarity of vocabulary-based representations as a proxy for method-comment similarity.

The first step in the similarity computation is source text processing. For text in comment parts it means identifying full period-terminated sentences using the Stanford CoreNLP toolkit [13], in case the comment consists of more than one sentence. Next, for each sentence we split all source code identifiers present into their individual constituents and expand all detected abbreviations. We selected an existing list of common English abbreviations, and extended it with widely-known abbreviations used in IT and Java projects. Our custom abbreviation expansion list can be straightforwardly substituted by other expansion lists, such as those from the dataset of Newman *et al.* [14]. Finally, we reduce each word to its stem, and we filter out common English stop words using the "Default English stopwords list."[7] After this step we transform the resulting text into a bag-of-words (BoW) representation. For the method signatures the pre-processing steps are similar, though in this case we start directly with identifier splitting.

After we have obtained two bag-of-words representations, we evaluate their similarity based on the occurrence of common words, for which we employ the *cosine similarity* measure. For a pair of BoWs we consider them to be related if the similarity measure value is above a threshold of 0.25 (MIN-THRESHOLD

[7] `https://www.ranks.nl/stopwords`

value in the Algorithm 1), on a scale from 0 (no similarity at all) to 1 (exact similarity).

Clone severity computation. After computing the similarity scores, RepliComment assigns a degree of severity to the issue (LOW, MILD, or HIGH) as described previously. Finally, RepliComment exports the results of its evaluation to a text (.txt) report file with a separate entry for each issue category. Each file reports:

1. the record in the csv file of clone suspects
2. the specific Java class the clone is from
3. a description of the issue(s) encountered
4. fix suggestions, which differ depending on the type of issue:
   (a) in the case of a HIGH severity issue, RepliComment points out which field or method is the one more related to the cloned comment, suggesting to fix the documentation of the other field or method
   (b) in the case of a MILD severity issue, RepliComment warns the user that the comment cloned across different fields or methods seems too generic, hence suggesting to fix each comment by providing more detail
   (c) in the case of a LOW severity issue, RepliComment warns the user of the clone found, but specifies that she may want to ignore the issue because it is likely a false positive (legitimate clone)

A portion of the txt file reporting HIGH severity issues looks like listing 1:

Listing 1: RepliComment Results file example

```
1  ----- Record #53 file:2020_JavadocClones_log4j.csv -----
2  In class: org.apache.log4j.lf5.LogRecord

4  1) The comment you cloned:"(@return)The LogLevel of this record."
5  seems more related to <LogLevel getLevel()> than <Throwable
6  getThrown()>

8  It is strongly advised to document method <Throwable getThrown()> with
9  a different, appropriate comment.

11 ----- Record #152 file:2020_JavadocClones_hadoop-hdfs.csv -----
12 In class: org.apache.hadoop.hdfs.util.LightWeightLinkedSet

14 1) The comment you cloned:"(@return)first element"
15 seems more related to <T pollFirst()> than <List pollN(int n)>

17 It is strongly advised to document method <List pollN(int n)> with
18 a different, appropriate comment.
```

4. Evaluation

In our evaluation we aim to understand the accuracy of RepliComment in *identifying and categorizing* comment clone issues. We also conduct a qualitative

Table 1: Subjects used for the evaluation of RepliComment. For each subject we report the number of implemented classes, the lines of Java code and the stars on GitHub as of July 2020

Project	Classes	LOC	Github ★
elasticsearch-6.1.1	2906	300k	50k
hadoop-common-2.6.5	1450	180k	11k
vertx-core-3.5.0	461	48k	11k
spring-core-5.0.2	413	36k	38k
hadoop-hdfs-2.6.5	1319	262k	11k
log4j-1.2.17	213	21k	718
guava-19.0	469	70k	38k
rxjava-1.3.5	339	35k	43k
lucene-core-7.2.1	825	103k	4k
solr-7.1.0	501	50k	4k
Total	1665	1105k	

analysis of the results to investigate whether the issues reported as HIGH severity, which are supposed to be the most worrisome comment clones, are indeed critical documentation issues that developers should fix. Finally, we compare the clone issues reported by RepliComment and by a code clone detection tool to study the correlation between code and comment clones.

For our empirical evaluation we select and analyze 10 projects among the most popular and largest repositories on GitHub, as listed in Table 1. Specifically, in our study we include projects developed in Java, since RepliComment targets this programming language, and these projects include a considerable number of classes documented with Javadoc. We selected these projects because they belong to different companies and developers (*e.g.*, Google, Apache, Eclipse), and thus the study is not biased towards specific documentation styles.

4.1. Evaluation Protocol and Research Questions

We resort to the official GitHub API[8] to obtain the source code of each subject listed in Table 1. For each project repository we run RepliComment on its source code to identify comment clones of different severity and category, and then further examine the results *manually* to assess their quality.

The manual analysis of the results involves the output of the *Clone detector*, as described in subsection 3.2, as well as the output of the *Clone analyzer* described in subsection 3.3, which reports the comment clones that deserve the developer's attention and classifies them by different severity levels. We analyze the intermediate output of the *Clone detector* to evaluate its ability to discard *legitimate* cases and discerning them from comment clones that deserve further analysis, the *non-legitimate* cases. We look into the final output, instead, to evaluate the ability of RepliComment to correctly classify comment clones.

Note that both outputs contain a high number of comment clones, as we will show in later sections. For this reason, we conduct our manual inspection on

[8] `https://developer.github.com/v3/`

random *samples* of cases. To randomly select a sample to evaluate manually, we grep all the Record # lines, such as lines 1 and 11 in listing 1, and then shuffle the desired number via the shuf GNU core utility. Details on the sizes of our samples follow in the respective answers to the research questions.

We now outline the research questions of our study.

- *RQ1: Are comment clones prevalent in popular Java projects?*

  We perform a quantitative study on all the classes of all the projects listed in Table 1 to motivate this work. We report the numbers of HIGH, MILD and LOW severity cases that we find in each subject, and we report the results in subsection 4.2.

- *RQ2: How accurate is RepliComment at differentiating* legitimate *and* non-legitimate *comment clones?*

  It is essential that RepliComment be able to differentiate between clones that developers should analyze and fix (*non-legitimate* clones), and clones that are *legitimate*. We manually analyze 225 samples of the HIGH, MILD and LOW severity cases that RepliComment reports as *non-legitimate* to assess whether they are false positives. Moreover, we manually analyze 200 samples among the cases that RepliComment flags as *legitimate* to assess if they are false negatives. We report the results of this evaluation in subsection 4.3.

- *RQ3: How effective are the newly-introduced heuristics at filtering our* legitimate *cases?*

  RepliComment-V1 [5] did not include all the heuristics and further improvements that we now implement. We evaluate how effective they are at reducing the number of false positives against the RepliComment-V1 implementation, and we present these results in subsection 4.4.

- *RQ4: How accurate is RepliComment at classifying the severity of* non-legitimate *comment clones?*

  We examine the manually analyzed samples of the previous research question, focusing on how accurate RepliComment is at flagging HIGH, MILD and LOW severity cases as such. The results of this evaluation appear in subsection 4.5

- *RQ5: Can RepliComment correctly identify the cloned vs. the original comment?*

  When RepliComment finds an instance of a *non-legitimate* comment clone due to a copy-paste error, it reports which comment of the pair is the one that should likely be fixed. We evaluate how accurate this information is in subsection 4.6.

- *RQ6: To what extent do comment clones detected by RepliComment correlate with code clone issues?*

Table 2: Quantitative results of the **method** comment clones reported by RepliComment on each analyzed project.

Project	Low		Mild		High		Tot. issues	Legit
	CP	WC	CP	WC	CP	WC		
elasticsearch	111	0	23	567	30	184	**915**	2221
Hierarchy	+4	+39	+2	+21	0	+6	+**72**	+51
Inter-class	+924	+28857	+138	+82	+117	+899	+**31017**	+4323
hadoop-common	100	0	75	173	28	4	**380**	3859
Hierarchy	+2	+15	0	0	0	+1	+**18**	+97
Inter-class	+64	+84	+569	+17	+55	+6	+**795**	+2314
vertx-core	33	0	139	53	795	4	**1024**	17433
Hierarchy	0	+1	+2	0	0	+3	+**6**	+378
Inter-class	+368	+115	+1636	0	+5579	+13	+**7711**	+109558
spring-core	46	0	78	83	15	6	**228**	2089
Hierarchy	+1	0	0	+3	+1	0	+**5**	+75
Inter-class	+192	0	+5	+8	+11	0	+**216**	+964
hadoop-hdfs	23	0	184	13	7	13	**240**	1198
Hierarchy	+1	+11	+12	0	+1	+1	+**26**	+71
Inter-class	+19	+608	+1131	+10	+12	+3	+**1783**	+897
log4j	1	0	3752	437	1	18	**4209**	16689
Hierarchy	0	+2	0	0	0	0	+**2**	+1434
Inter-class	+16	+6	+3752	+9	+1	+4	+**3788**	+18615
guava	75	0	63	215	77	63	**493**	1122
Hierarchy	+2	+1	+127	+44	0	+4	+**178**	+79
Inter-class	+16	+9	+2066	+49	+20	+6	+**2166**	+4091
rxjava	3558	0	12	15	48	4	**3637**	11533
Hierarchy	0	0	0	0	0	0	**0**	0
Inter-class	+2	+3	+13	+12	+5	0	+**35**	0
lucene-core	25	0	84	65	1	50	**225**	1062
Hierarchy	+5	+6	+4	0	0	+2	+**17**	+295
Inter-class	+345	+118	+516	+710	+6	+46	+**1741**	+4268
solr	1	0	3	9	2	2	**17**	4253
Hierarchy	0	+1	0	0	0	0	+**1**	+14
Inter-class	0	0	0	+2	+1	0	+**3**	+689
Total,Intra-class	3973	0	4413	1630	1004	348	**11368**	61459
Additional,Hierarchy	15	76	147	68	2	17	**325**	2494
Additional,Inter-class	1946	29800	9826	899	5807	977	**49255**	145719

We investigate how often RepliComment reports comment clone issues for methods that are detected as clones by code clone detection tools, and report our findings in subsection 4.7.

The following subsections present our answers to the research questions. Overall, we manually analyze over *500 cases* of comment clones.

4.2. RQ1: Prevalence of Comment Clones

Table 2 shows the complete quantitative data that RepliComment outputs for the *method* comment clone search. We report the number of comment clones by type of clone (CP — comment part, WC — whole comment) and severity of the issue (Low, Mild or High). For each project, the first row reports the results of running RepliComment with default scope search (*i.e.*, Intra-class); the second row (Hierarchy) reports the additional clones with class hierarchy scope; and the last row (Inter-class) the additional clones with Inter-class search scope.

RepliComment reports a total of 11,368 method comment clones considered to be potential issues, and discards 61,459 comment clones considered to be *legitimate*. For the hierarchy search, RepliComment reports 325 additional potentially harmful clones, while it flags 2494 additional *legitimate* clones. Finally, for the inter-class search, RepliComment reports 49,255 additional clones, while 145,719 more clones are labeled as *legitimate*.

legitimate We can see that the vast majority of the comment clones are not harmful. The total of 209,672 comment clones labeled as *legitimate* by the *Clone detector* heuristics are not subsequently analyzed by the *Clone analyzer*, and therefore are not reported to developers. 60,948 are left to be analyzed, namely, 23% of the total reported issues.

LOW In the intra-class search, 3,973 cases, *i.e.*, 35% of the 11,368 *non-legitimate* reported issues, are considered to be LOW severity issues, and they all come from comment part clones. In hierarchy search, this is the case for 91 cases of 325 (or 28%), 15 for comment part clones and 76 for whole comment clones. For inter-class search, 31746 (1946 comment parts, 29,800 whole comments) are LOW severity issues over a total of 49,255 (or 64%). This means that the *Clone analyzer* component of RepliComment thinks all those cases might be false positives, despite overcoming the filtering heuristics of the *Clone detector* (subsection 3.2). Thus, RepliComment is able to prune additional clones thanks to the analysis phase.

MILD In the intra-class search, 53% of the 11,368 issues, consisting of 4,413 clones of comment parts, and 1,630 clones of whole comments, are considered to be MILD severity issues by RepliComment. The same applies in the hierarchy search in 66%, and in inter-class search in 22% of the times, respectively.

This means that large proportions of problematic comment clones are considered to be due to poor information quality in the documentation. This is not surprising to us, as our initial hypothesis was that code comment clones are mostly due to lack of proper information rather than oblivious copy-and-paste errors.

HIGH Finally, in the intra-class search, RepliComment reports that 12% of the 11,368 issues, consisting 1,004 cases of clones in comment parts and 348 cases of whole comment clones, are HIGH severity issues. In the hierarchy search this happens only for a small proportion of 6% of cases, and, in the inter-class search, of 14% of cases. Overall, 8155 cases over a total of 60,948 analyzed ones (13%) are considered to be HIGH severity issues. These are the issues that RepliComment considers to need an urgent fix.

Table 3 shows all clones that RepliComment reports for *field* comment clones. Since field comments have no tags, there is no distinction between comment parts and whole comment clones.

In the intra-class search, RepliComment reports a total of 44 field comment clones considered to be potential issues, while none is considered *legitimate* right away. In the hierarchy search, RepliComment reports no additional potentially harmful clones, while it flags only 2 additional *legitimate* clones. Finally, in the inter-class search, RepliComment reports 9 additional problematic clones, while it labels 134 additional ones as *legitimate*. The overall number of potential issues is 53:

Table 3: Quantitative results of the **field** comment clones reported by RepliComment on each analyzed project.

Project	Low	Mild	High	Tot. issues	Legit
elasticsearch	2	1	0	**3**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+19
hadoop-common	1	21	0	**22**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	+1	0	+**1**	+6
vertx-core	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	+2	+1	0	+**3**	+14
spring-core	6	0	0	**6**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+7
hadoop-hdfs	1	3	1	**5**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+4
log4j	0	3	0	**3**	0
Hierarchy	0	0	0	**0**	+2
Inter-class	+1	0	0	**0**	+65
guava	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+6
rxjava	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+3
lucene-core	1	4	0	**5**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	4	0	+**4**	+10
solr	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	0
Total, Intra-class	11	32	1	**44**	0
Additional, Hierarchy	0	0	0	**0**	+2
Additional, Inter-class	+3	+6	0	+**9**	+134

Low A total of 14 issues, hence 26% of the total, are considered of Low severity.

Mild Most of the issues, *i.e.*, 38 (72% of the total), are considered to be of Mild severity, hence providing poor information.

High Only a single issue is considered to be a High severity one, and it is detected through an intra-class search.

Given the results of this experiment, we conclude that comment clones are prevalent even in popular Java projects. The results of the search with different scopes seem to show that RepliComment should better be used either with Intra-class or Hierarchy scopes, as looking for comment clones with Inter-class scope reports too many method comment clones to be analyzed by developers, despite the ability of RepliComment to filter out many legitimate cases.

4.3. RQ2: Accuracy of RepliComment at differentiating legitimate *and* non-legitimate *clones*

We *manually* analyze some samples of the clones that RepliComment identifies as *legitimate* or not to establish the rate of false positives and false negatives. We first present the results regarding method comments, separating clones of comment parts and whole comment clones. We then proceed with the results of field comments.

Table 4: Clones of comment parts and whole comments after duplicate removal

	Comment part clones				Whole comment clones			
Project	Low-CP	Mild-CP	High-CP	Total	Low-WC	Mild-WC	High-WC	Total
elasticsearch	111	15	6	**132**	0	377	103	**480**
HIERARCHY	+4	+2	0	**+6**	+7	+2	+6	**+15**
INTER-CLASS	+461	+119	+33	**+613**	+2	+10	+503	**+515**
hadoop-common	34	34	13	**81**	0	0	0	**0**
HIERARCHY	+2	0	0	**+2**	+15	0	+1	**+16**
INTER-CLASS	+64	+221	+24	**+309**	+84	+3	+6	**+93**
vertx-core	27	15	14	**56**	0	13	6	**19**
HIERARCHY	0	+2	0	**+2**	+1	0	+3	**+4**
INTER-CLASS	+368	+13	+2	**+383**	+3	0	+1	**+4**
spring-core	46	20	12	**78**	0	46	20	**66**
HIERARCHY	+1	0	+1	**+2**	0	+3	0	**+3**
INTER-CLASS	+36	+5	+11	**+52**	0	+8	0	**+8**
hadoop-hdfs	23	28	7	**58**	0	13	11	**24**
HIERARCHY	+1	+12	+1	**+14**	+11	0	+1	**+12**
INTER-CLASS	+19	+895	+12	**+926**	+6	+10	+3	**+19**
log4j	1	1	1	**3**	0	15	3	**18**
HIERARCHY	0	0	0	**0**	+2	0	0	**+2**
INTER-CLASS	+16	+1	+1	**+18**	+6	+9	+4	**+19**
guava	57	24	9	**90**	0	132	48	**180**
HIERARCHY	+2	+127	0	**+129**	+1	+1	+4	**+6**
INTER-CLASS	+16	+7	+9	**+32**	+9	+39	+6	**+54**
rxjava	23	7	3	**33**	0	15	2	**17**
HIERARCHY	0	0	0	**0**	0	0	0	**0**
INTER-CLASS	+2	+13	+5	**+20**	+3	+1	0	**+4**
lucene-core	25	21	1	**47**	0	65	24	**89**
HIERARCHY	+5	+4	0	**+9**	+6	+2	0	**+8**
INTER-CLASS	+345	+516	+6	**+867**	+25	+6	+16	**+47**
solr	1	3	2	**6**	0	9	2	**11**
HIERARCHY	0	0	0	**0**	+1	0	0	**+1**
INTER-CLASS	0	0	+1	**+1**	0	+2	0	**+2**
Total,INTRA-CLASS	1690	2105	174	**3969**	182	781	773	**1736**
Additional,HIERARCHY	15	147	2	**164**	44	8	15	**395**
Additional,INTER-CLASS	1327	1790	104	**3221**	138	88	539	**7207**

4.3.1. Method comment clones

False positives. We manually inspect all the entries in table 2 to ensure a fair sampling, and we remove duplicates to ensure that sampling catches the largest variety of comments. For this purpose, we consider a case to be a duplicate if the comment is exactly the same, but affects multiple method instances. This is likely to happen when developers write generic `@throws` comments such as *"on error"* for all the documented exceptions, for instance. Note that we draw this distinction for manual analysis, but in reality comment clones affecting multiple methods should all be addressed by developers.

Table 4 lists the unique comment clone instances after duplicates removal, reporting comment part clones and whole comment clones separately.

We sample entries of table 4 by selecting *at least* 10% of the cases for each category (LOW, MILD, HIGH for intra-class, hierarchy and inter-class search). We sample 225 issues for intra-class, 63 for hierarchy, and 124 for inter-class search, for a total of 412 issues.

Regarding **intra-class search**, we find:

- For *comment parts*, we have 50 MILD issues and 30 HIGH issues. We disagree on a total of 33 issues, 26 MILD and 7 HIGH. In particular, all 7 HIGH issues are false positives, so such clones are actually legitimate. Among the 26 MILD cases, 22 of them are false positives (the rest should have been considered HIGH severity issues). Thus RepliComment produces **29 false positives** for clones of comment parts.
- For *whole comment* clones, we have 70 MILD issues and 25 HIGH issues.

We disagree on a total of **12** issues, 10 MILD and 2 HIGH, and **all of them are false positives**. A common reason why whole clones of comments can still be considered *legitimate* is that an API class is not supported anymore, and its method documentation states so (advising to avoid using the method and pointing to another class, *etc.*).

- In conclusion, RepliComment reports 45 false positives for a total of 175 samples for intra-class search, which suggests a precision of 74% of RepliComment in intra-class search.

Regarding **hierarchy search**, we have:

- For *comment parts*, we never disagree with RepliComment in the additional sampled 17 issues (15 MILD and 2 HIGH ones).
- For *whole comment* clones, we never disagree on the assessment made on 10 MILD, while we do disagree for 11 HIGH ones.
- In conclusion, RepliComment achieves a precision of 71% for hierarchy search.

Listing 2 show an example of a HIGH-severity comment part clone found while exploring a class hierarchy. The same clone was found during an intra-class search (see listing 1): Bad clones existing in one class may be replicated in its subclasses, thus perpetuating the issue.

Listing 2: Hierarchy high-severity issue (RepliComment report)

```
1 ----- Record #4 file:2020_JavadocClones_h_hadoop-hdfs.csv -----
2 In class: org.apache.hadoop.hdfs.util.LightWeightLinkedSet
3 And its superclass: org.apache.hadoop.hdfs.util.LightWeightHashSet

5 1) The comment you cloned:"(@return)first element"
6 seems more related to <T pollFirst()> than <List pollN(int n)>
```

Finally, for **inter-class search**, we have that:

- For *comment parts*, we disagree with 4 RepliComment assessments over a total of 31 (16 MILD and 15 HIGH).
- For *whole comment* clones, we disagree with 2 assessments over a total of the 65 (10 MILD and 55 HIGH) issues sampled.
- In conclusion, RepliComment reports 6 false positives over a total of 96 issues, achieving a precision of 94%.

As an example, consider Listing 3. The interesting fact is that the two different classes across which the whole comment was cloned are not in the same hierarchy, and in general have little in common: they do not even belong exactly to the same package.

Listing 3: Inter-class high-severity issue (RepliComment report)

```
1 ----- Record #6 file:2020_JavadocClones_cf_hadoop-hdfs.csv -----
2 In class: org.apache.hadoop.hdfs.tools.offlineEditsViewer.XmlEditsVisitor
3 And class:
4 org.apache.hadoop.hdfs.tools.offlineImageViewer.TextWriterImageVisitor

6 You cloned the whole comment for methods
7 < XmlEditsVisitor(OutputStream out)> and
```

```
 8 | < TextWriterImageVisitor(String filename, boolean printToScreen)>

10 | The comment you cloned:"(Whole)Create a processor that writes to the
11 | file named and may or may not also output to the screen, as specified.
12 | @param Name of file to write output to @param Mirror output to screen?"
13 | seems more related to <TextWriterImageVisitor(String filename, boolean
14 | printToScreen)> than <XmlEditsVisitor(OutputStream out)>
```

False negatives. Our heuristics could wrongly flag as *legitimate* some clones that actually represent real issues. Cases marked as *legitimate* are filtered out in the first phase, *i.e.*, they are not analyzed further. Thus, in the case of a false negative, the issue would never be revealed. It is hence important to check that false negatives are not pervasive.

RepliComment marks as *legitimate* the comment clones reported in Table 2. We do not distinguish between comment parts and whole comments because a whole comment clone can never be considered *legitimate*.

We randomly sample 20 cases for each project and each type of search. If the total number is less than 20 then we analyze all cases. We manually analyze each of the *572 comment clones* to check whether it should indeed be considered to be *legitimate* (*i.e.*, we agree with RepliComment heuristics) or *non-legitimate* (*i.e.*, it is a false negative).

Table 5: Total of clones considered legitimate by the heuristics

Project	Agree (legit)	Disagree (non-legit)	Precision
elasticsearch-6.1.1	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
hadoop-common-2.6.5	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
vertx-core-3.5.0	20	0	100%
Hierarchy	19	1	95%
Inter-class	20	0	100%
spring-core-5.0.2	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
hadoop-hdfs-2.6.5	19	1	95%
Hierarchy	20	0	100%
Inter-class	20	0	100%
log4j-1.2.17	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
guava-19.0	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
rxjava-1.3.5	20	0	100%
Hierarchy-	-	-	-
Inter-class	20	0	100%
lucene-core-7.2.1	20	0	100%
Hierarchy	20	0	100%
Inter-class	20	0	100%
solr-7.1.0	20	0	100%
Hierarchy	14	0	100%
Inter-class	20	0	100%
Total	572	2	99.7%

Table 5 shows that we disagree with the classification as *legitimate* in two comment clones over 572 randomly selected in total. This means that we find *only two false negatives in our random sampling.* In particular, one is a case of a very generic exception comment that RepliComment's heuristics miss. The second is the case of parameters documented with the same name (for which a comment clone is tolerated), having however, different non-primitive types.

4.3.2. Field comments

False positives. Since RepliComment reports a relatively low number of issues for field comments, namely 38 MILD and only one HIGH, we analyze them all. Most of the MILD severity issues, namely 21, are all from hadoop-common. These clones would probably be considered legitimate by developers, since the comment states: *"This constant is accessible by subclasses for historical purposes. If you don't know what it means then you don't need it."* Hence, we consider these instances to be false positives. We also flag as false positives 3 instances from hadoop-common: in this case, field names are not parsable correctly due to multiple words being merged into a single one (*e.g.*, DFS_DATATRANSFER_SERVER_VARIABLEWHITELIST_FILE). We agree with the remaining 14 MILD ones, as well as with the single HIGH severity issue (see Listing 4). This suggests a precision of 39%.

Listing 4: Only high-severity issue existing for field clones (RepliComment report)

```
2 ---- Record #7 file:2020_JavadocClones_fields_hadoop-hdfs.csv ----
3 In class: org.apache.hadoop.hdfs.shortcircuit.ShortCircuitCache


6 1) The comment you cloned:"(Field)The executor service that runs the
7 cacheCleaner."
8 seems more related to <cleanerExecutor> than <releaserExecutor>
```

Listing 4 shows the only HIGH-severity issue RepliComment finds when exploring field clones, along with its assessment. The clone exists within the same class.

False negatives. We sample 20 instances from the 136 total *legitimate* field-level clones, and we confirm that we do agree with all of RepliComment's assessments.

This analysis shows that RepliComment's heuristics can be trusted to filter out many legitimate comment clones, and the rate of false positives is acceptable for practical use.

4.4. RQ3: Improvement of Heuristics over RepliComment-V1

We assess how well new heuristics implemented in the clone detector filter out further false positives in RepliComment compared to RepliComment-V1. To compare the effectiveness of the heuristics, we take the intersection of comment clones that RepliComment-V1 and RepliComment identify, and we compare their classification results. Table 6 presents the percentage of clones that RepliComment-V1 and RepliComment report as non-legitimate. The ability to report *fewer* issues is positive given the fact that in subsection 4.3 we assessed that heuristics do not cause false negatives. The table highlights the following results:

- In half of the projects (marked in bold font) the decrease of clones marked as *non-legitimate* by the heuristics is significant, going from a minimum reduction of -7% (spring-core-5.0.2) to a maximum of -29% (vertx-core-3.5.0);

Table 6: Samples of clones marked as non-legitimate before and after new heuristics application

Project	Old heuristics	New Heuristics
elasticsearch-6.1.1	49%	29%
hadoop-common-2.6.5	10%	9%
vertx-core-3.5.0	35%	6%
spring-core-5.0.2	17%	10%
hadoop-hdfs-2.6.5	9%	17%
log4j-1.2.17	20%	20%
guava-19.0	31%	31%
rxjava-1.3.5	38%	24%
lucene-core-7.2.1	19%	18%
solr-7.1.0	16%	0.5%
Average	24%	16%

- In four projects the reduction was close to non-existent, which means that some false positives are potentially retained, but no new ones are introduced;
- In only one project (hadoop-common-2.6.5) did the number of clones marked as *non-legitimate* increase by +8% instead of diminishing, potentially leading to an increase in the number of false positives.

4.5. RQ4: Accuracy of RepliComment at Classifying legitimate *Comment Clones*

We manually evaluate RepliComment's assessment for each entry in the samples to determine its accuracy at classifying HIGH, MILD and LOW clones. Results report if our manual evaluation agrees or disagrees with RepliComment's assessment. If we disagree, it means that RepliComment assigns the wrong category to one case, for example reporting it as a MILD severity when it is actually a LOW one. Conversely, if we agree it means we would assign the same level of severity to the case.

Method-level analysis. Overall, we manually inspect and assess *412* reported issues. Table 7 reports the analysis for clones of comment parts. Results show that:

- RepliComment is very effective at classifying both LOW (>80%) and HIGH (>70%) severity issues in all kinds of search (intra-class, hierarchy, inter-class). This means RepliComment can highlight the most critical clones (copy-paste issues) that developers should focus on.
- On the other hand, RepliComment often fails at identifying MILD severity issues as such, since RepliComment analysis fails nearly half of the times during intra-class search. We carefully analyzed the wrong classifications to give an explanation to this discrepancy: it appears to be a problem of linguistic *semantics*. RepliComment, in the current implementation, is

Table 7: Manual analysis of RepliComment assessment for clones of Javadoc parts (summary, @param, @return or @throws)

Category		Sample	Agree	Disagree	Precision
Intra-class	Low-CP	50	42	8	84%
	Mild-CP	50	24	26	48%
	High-CP	30	23	7	77%
Hierarchy	Low-CP	15	15	0	100%
	Mild-CP	15	15	0	100%
	High-CP	2	2	0	100%
Inter-class	Low-CP	14	14	0	100%
	Mild-CP	16	16	0	100%
	High-CP	15	11	4	73%
Total		207	162	45	
Average precision					87%

neither aware of synonyms nor particular developer jargon. For example, our manual analysis reveals that oftentimes developers refer to a primitive parameter (being it int, long, char, *etc.*) generically as *"the value"*. RepliComment's bag of words representations do not map such an expression to any portion of the method signature, since typically parameters have a specific name and type that differ from *"value"*. Hence, the analysis concludes that the cloned comment does not relate enough either to the first method or to the second one, maybe because it is too generic. Unfortunately such cases are false positives (Low severity). By tackling synonyms correctly, RepliComment would not report as an issue most of the wrongly classified cases.

Table 8 reports the analysis for clones of whole comments:

Table 8: Manual analysis of RepliComment assessment for whole Javadoc clones

Category		Sample	Agree	Disagree	Precision
Intra-class	Low-WC	0	0	0	0%
	Mild-WC	70	60	10	86%
	High-WC	25	23	2	92%
Hierarchy	Low-WC	10	10	0	100%
	Mild-WC	10	10	0	100%
	High-WC	11	0	11	0%
Inter-class	Low-WC	14	14	0	100%
	Mild-WC	10	10	0	100%
	High-WC	55	53	2	96%
Total		205	180	25	
Average precision					75%

Precision of RepliComment in classifying both MILD and HIGH severity issues in all kinds of search for whole comment clones tends to be very high (∼90%), except for hierarchy search. In general, if a whole comment is copied for an overloaded method, it most likely means that the developer simply forgot to document the difference in the parameters, which would be a MILD severity issue. On the other hand, if a whole comment is copied across methods that are not overloaded, something is likely to be off. We report a particular example of this in Listing 5:

Listing 5: RepliComment HIGH severity whole comment clone example

```
1  ———— Record #519 file:2020_JavadocClones_elastic.csv ————
2  In class: org.elasticsearch.common.collect.ImmutableOpenMap
3  1) You cloned the whole comment for methods
4  <Iterator keysIt()> and
5  <Iterator valuesIt()>
6
7  This is not an overloading case. Check the differences among the two
8  methods and document them.
9
10 2) The comment you cloned:"(Whole)Returns a direct iterator over the
11 keys."
12 seems more related to <Iterator keysIt()> than <Iterator valuesIt()>
```

As for the hierarchy search, RepliComment misclassifies constructor comments. Overall, it reports a low number of HIGH severity issues, but unfortunately they all look like false positives. To properly tackle constructor comments, more advanced assessments may be needed.

Field-level analysis. We analyze 14 LOW-severity issues, 38 MILD-severity issues and only one HIGH-severity issue. We consider correct all LOW-severity issues, which include 11 clones identified during intra-class search, and 3 additional clones identified during inter-class search. Regarding MILD-severity issues, we believe 24 are wrongly classified, since they should probably be labeled as LOW. We consider correct the only HIGH-severity issue coming from an intra-class analysis of hadoop-hdfs. This yields a precision of 100% for LOW and HIGH severity issues, and of 39% for MILD severity issues.

The results of this experiment show that RepliComment is effective at differentiating comment clones, so developers can effectively focus on the most critical ones first.

4.6. RQ5: Ability to Identify Cloned and Original Comments

The ultimate goal of RepliComment is to support developers in pointing out *which* comment to fix, when the clone is due to a copy-and-paste error. In this section we evaluate how good RepliComment is at distinguishing the original and the cloned comment.

4.6.1. Method-level analysis

Intra-class clones. To answer this question, we examine RepliComment's assessment for the same 30 entries of HIGH-CP in Table 7, and the 25 HIGH-WC entries in Table 8.

- For HIGH-CP, we exclude the seven entries for which we disagree, since according to our manual inspection they are not real copy-paste issues. Our manual analysis confirms the correctness of RepliComment in pointing out the comment that was cloned for all the remaining 23 cases out of 30. Thus, the tool correctly suggests to the developer which method needs a documentation fix with a precision of 77%.

- Similarly, for HIGH-WC, we exclude the two entries for which we disagree. Our manual analysis reveals that we are unsure about three suggestions out of 23, and we do not agree with one out of 23 because we can infer that the two methods are actually equivalent in behavior (RepliComment in such a case should suggest that each of the methods is similarly related to the comment, meaning that neither of them appears better than the other). We completely agree with the suggestions for the remaining 19 out of 23 cases, which yields a precision of 83% in suggesting the right fix to the developer.

Hierarchy clones. We examine RepliComment's assessment for the two entries of HIGH-CP in Table 7 and the eleven HIGH-WC entries in Table 8.

- For HIGH-CP, we do agree with both RepliComment's picks. It is interesting to note that one is an example already found via intra-class analysis of hadoop-hdfs, which was replicated in the hierarchy.

- We exclude HIGH-WC, since we disagreed with all of their assessments.

Inter-class clones. We examine RepliComment's assessment for the 15 entries of HIGH-CP in Table 7 and the 55 HIGH-WC entries in Table 8.

- For HIGH-CP, we exclude the four instances for which we disagree with RepliComment. We do agree with all the remaining ones. Interesting examples of such clones can be found in section 2.

- Similarly, for HIGH-WC, we exclude two instances. As for the remaining 53 ones, it is worth noting that 49 of them seem to arise from the same elastic patterns of documentation. For example, the developers tend to write comments like *"Sets the minimum score below which docs will be filtered out"* both for actual setter methods and methods which are not actually setters, or at least, methods which perform some extra operations beside setting a value. Hence, RepliComment is justified in picking the setter method as the right owner of the comment. That said, those are probably voluntary habits accepted by the project's developers, and not actual copy-and-paste slips. Excluding such instances, we are left with four, which do look like oblivious copy-and-paste mistakes and for which we agree with RepliComment's pick.

Field-level analysis. As for field-level analysis, we only have a single instance of HIGH severity issue, for which we confirm the assessment of RepliComment.

This experiment confirms that RepliComment can actually support developers in highlighting which comments are the original ones and which ones are copied, and therefore should be fixed.

4.7. RQ6: Correlation with code clones

Comment clones may be the result of copy-and-paste practice on entire method implementations. If this was the case, comment clones would appear only when their corresponding method implementations are clones as well. To understand if this is the case, we compare clone issues reported by RepliComment and by NiCad 2.6 code clone detector [15]. We follow this comparison protocol for each of the projects:

- We extract class-qualified signatures of methods for which RepliComment reports HIGH severity issues in Javadoc comments for both comment parts and whole comments in all three analysis modes (within the same file, within the class hierarchy, and across all classes of the project);
- We extract class-qualified signatures of methods which NiCad reports as type III (near-miss blind renamed) clones with first over 70% and then only with exactly 100% similarity using the default configuration (clones sized between 10 and 2500 LOC, the near-miss difference threshold set to at most 30% different lines); We use the default code clone similarity threshold of NiCad clone detector as a baseline in our experiments. The difference of 30% is already quite liberal in the context of code clones, and previous studies on human judgment of code clones suggest that it is not trivial to agree on when a clone becomes a legitimate method with just a similar structure [16].
- We pipe GNU core utilities sort and comm to sort outputs of both tools and compare them line by line, respectively.

Additionally, we collect the statistics of how many methods reported as code clones by NiCad have Javadoc comments. Table 9 presents such data both for exact and non-exact code clones.

We can see from the statistics collected that code clones seem to be fairly well-documented, with a minimum percentage of commented methods of 15% in elasticsearch and a maximum percentage of 92% in hadoop-hdfs. The remaining eight projects can be further split into two groups, where in the first group the rate of documented code clones is around 30%, and in the other group this rate is closer to 60%. However, across the 10 projects we have detected only a few cases for which both RepliComment and NiCad tools reported clone issues in the same methods. RepliComment reported whole comment clones in the same file, the first clone tuple consisting of two methods in the rxjava project, and the second clone tuple of three methods in the lucene project, where both clone tuples consist of exact code clones (code similarity 100%). Additionally,

Table 9: Code clones statistics

Project	Code clones exact			Code clones 70%+ similar		
	All	Commented	Matching	All	Commented	Matching
elasticsearch-6.1.1	153	43 (28%)	0	1248	193 (15%)	**29**
hadoop-common-2.6.5	155	95 (61%)	0	1047	364 (34%)	0
vertx-core-3.5.0	23	6 (28%)	0	202	56 (27%)	0
spring-core-5.0.2	22	17 (77%)	0	143	89 (62%)	0
hadoop-hdfs-2.6.5	422	389 (92%)	0	5764	2093 (36%)	0
log4j-1.2.17	18	10 (55%)	0	90	40 (44%)	0
guava-19.0	84	37 (44%)	0	417	224 (53%)	**3**
rxjava-1.3.5	35	10 (28%)	**2**	332	102 (30%)	**2**
lucene-core-7.2.1	73	24 (32%)	**3**	592	175 (29%)	**3**
solr-7.1.0	129	25 (19%)	0	528	84 (16%)	0

when lowering code clone similarity threshold to 70% RepliComment and NiCad report matching issues in two additional projects: in the elasticsearch project 29 code clones distributed over 7 different clone classes with in-class similarity varying from 70% to 91% are also reported by RepliComment as methods with inter-class whole comment clones, and in the guava project 3 code clones distributed over 1 clone class with in-class similarity of 72% are also reported by RepliComment as methods with intra-class comment part clones.

Our findings indicate that critical comment clones issues cannot necessarily be well-detected by code clone detection tools, as in most cases the clones in comments were considered to be legitimate by RepliComment.

5. Related work

The works by Oumaziz *et al.* [17] and Luciv *et al.* [18] study what we call *legitimate* clones to encourage smart documentation reuse. Despite the different scope of these works compared to RepliComment, some of their findings are relevant for our research as well. In particular, Luciv *et al.* [18] highlight that exact documentation clones are by far the most common, and that near-duplicate detection techniques still carry many false positives.

Considerable work on clone detection focuses on *code* clones [11]. Typically, code clone detection techniques remove comments and whitespace from the source code to eliminate spurious information [19, 20, 11]. Indeed, considering comments while searching for code clones could lead to missing some relevant code clones that differ only in their comment descriptions. The work by Marcus *et al.* is an exception to this practice [21]. Their code clone detection technique actually performs better with comments, since comments carry relevant information, as the authors themselves acknowledge. Marcus *et al.* however, do not report comment clones per se, as RepliComment does. Mayrand *et al.* also recognize the value of code comments, since metrics such as code volume identify similar layouts (*i.e.*, possible code clones) inside the source code, and comments help in this respect [22]. Nonetheless, the aim of our work is different from general code clone detection. The Javadoc clones that RepliComment reports typically belong neither to similar nor equal method implementations. The problem we tackle is actually the opposite: two methods, with properly

different implementations, may erroneously have the same comment because it was copied and pasted from another method.

Our long term aim is to address low quality documentation issues, and some previous work exists. Steidl *et al.* have some purposes in common with our work [23]. They study techniques to assess the coherence between comments and code. They compare the lexical similarity of comments and code to verify if the same terms are used, with an *edit distance* of 2. Their work could identify some copy-paste issues. However, most of the *legitimate* clones we found in our experiment would be wrongly reported as *non-legitimate* by their technique. We believe this problem can be addressed more precisely, for example, via a more comprehensive semantic analysis. Khamis *et al.* developed JavadocMiner [24], a tool that assesses the overall quality of Javadoc comments. They measure comment quality using classical NLP metrics (such as the readability index). However their main purpose is to verify that the Javadoc standard is correctly used, *e.g.*, a `@param` tags comment should start with the name of the documented parameter. Another relevant work on comment quality by Zhong *et al.* [25] focuses on detecting syntax errors and broken code names. These techniques nicely complement RepliComment.

6. Conclusions and Future Work

The purpose of our work is to help developers to identify and fix issues in code documentation. We started working in this direction by focusing on comment clones. We have implemented RepliComment, a prototype to automate the identification and classification of source comment clones that may be worthy of attention.

As future work we foresee many tasks. First and foremost, we aim to introduce new heuristics to better classify comment clones. Secondly, we plan to further automate the analysis after the classification of a comment clone. In the presence of copy-paste issues, for instance, we could not only automatically identify which method is the source, and thus which comment should be fixed by developers, but also improve the precision of our report, and present the cloned part to a developer with a concrete fix suggestion.

We could employ natural language analysis on the cloned comment and their corresponding method signatures, and report the mismatching cases. There are various techniques in the state of the art to assess document similarities, such as Word Embedding [26]. We could compare the semantics of method names to the semantics of their corresponding comments. We would report as likely to fix the comment clones for which the method name is less similar to the comment.

The analysis for "poor information" clones could benefit from additional metrics. There exist various metrics to assess text characteristics, such as its complexity, its quality, and the quantity of information it describes. We could integrate these metrics into RepliComment to improve its ability to classify comment clones.

Last but not least, we would like RepliComment to be properly integrated into an IDE to automatically notify developers while they write code, and flag

corresponding comments with warning messages such as "This comment seems to belong to method X, and not to method Y. Verify this clone and correct the comment for method Y if necessary", or "This comment includes generic information. Please provide a better description."

References

[1] A. Goffi, A. Gorla, M. D. Ernst, M. Pezzè, Automatic generation of oracles for exceptional behaviors, in: ISSTA 2016: Proceedings of the 2016 International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 2016, pp. 213–224. doi:10.1145/2931037.2931061.

[2] S. H. Tan, D. Marinov, L. Tan, G. T. Leavens, @tComment: Testing Javadoc comments to detect comment-code inconsistencies, in: ICST 2012: 5th International Conference on Software Testing, Verification and Validation, Montreal, Canada, 2012, pp. 260–269.

[3] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, F. Qin, Automatic model generation from documentation for Java API functions, in: ICSE 2016: Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 2016, pp. 380–391.

[4] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, H. Gall, Analyzing APIs documentation and code to detect directive defects, in: ICSE 2017: Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, 2017, pp. 27–37.

[5] A. Blasi, A. Gorla, RepliComment: Identifying clones in code comments, in: ICPC 2018: Proceedings of the 26th IEEE International Conference on Program Comprehension, Gothenburg, Sweden, 2018, pp. 320–323.

[6] A. Corazza, V. Maggio, G. Scanniello, Coherence of comments and method implementations: a dataset and an empirical investigation, SQJ (2016) 1–27.

[7] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. Gueheneuc, G. Antoniol, Physical and conceptual identifier dispersion: Measures and relation to fault proneness, in: ICSM 2010: 26th IEEE International Conference on Software Maintenance, Timişoara, Romania, 2010, pp. 1–5.

[8] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, D. C. Shepherd, Software documentation: The practitioners' perspective, in: ICSE 2020: Proceedings of the 42nd International Conference on Software Engineering, Seoul, Republic of Korea/Virtual, 2020.

[9] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, K. A. Schneider, CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API

Documentation, in: ASE 2019: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, California, USA, 2019, pp. 1026–1037.

[10] K. W. Nafi, B. Roy, C. K. Roy, K. A. Schneider, CroLSim: Cross Language Software Similarity Detector Using API Documentation, in: SCAM 2018: Proceedings of the 18th International Working Conference on Soource Code Analysis and Manipulation, Madrid, ICSME, 2018, pp. 139–148.

[11] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Tech. Rep. 2007-541, Queen's University, School of Computing (2007).

[12] N. Stulova, A. Blasi, A. Gorla, O. Nierstrasz, Towards detecting inconsistent comments in Java source code automatically, in: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, 2020, pp. 65–69.

[13] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, D. McClosky, The Stanford CoreNLP natural language processing toolkit, in: ACL 2014: The 52nd Annual Meeting of the Association for Computational Linguistics, Baltimore, USA, 2014, pp. 55–60.

[14] C. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, E. Hill, An open dataset of abbreviations and expansions, in: ICSME 2019: 35th IEEE International Conference on Software Maintenance and Evolution, Cleveland, OH, USA, 2019, pp. 280–280.

[15] J. R. Cordy, C. K. Roy, The NiCad clone detector, in: ICPC 2011: Proceedings of the 19th IEEE International Conference on Program Comprehension, Kingston, ON, Canada, 2011, pp. 219–220.

[16] C. Kapser, P. Anderson, M. W. Godfrey, R. Koschke, M. Rieger, F. V. Rysselberghe, P. Weißgerber, Subjectivity in clone judgment: Can we ever agree?, in: R. Koschke, E. Merlo, A. Walenstein (Eds.), Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006, Vol. 06301 of Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[17] M. A. Oumaziz, A. Charpentier, J.-R. Falleri, X. Blanc, Documentation reuse: Hot or not? an empirical study, in: ICSR 2017: 16th International Conference on Software Reuse, Salvador, Brazil, 2017, pp. 12–27.

[18] D. V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Y. Romanovsky, D. A. Grigoriev, Detecting near duplicates in software documentation, Program. and Comp. Soft. 44 (5) (2018) 335–343.

[19] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE TSE 28 (7) (2002) 654–670. doi:10.1109/TSE.2002.1019480.

[20] J. Krinke, A study of consistent and inconsistent changes to code clones, in: WCRE 2007: 14th Working Conference on Reverse Engineering, Vancouver, BC, Canada, 2007, pp. 170–178. doi:10.1109/WCRE.2007.7.

[21] A. Marcus, J. I. Maletic, Identification of high-level concept clones in source code, in: ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering, Lawrence, KS, USA, 2011, pp. 107–114. doi:10.1109/ASE.2001.989796.

[22] J. Mayrand, C. Leblanc, E. M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: ICSM '96: Proceedings of the International Conference on Software Maintenance, Monterey, CA, USA, 1996. doi:10.1109/ICSM.1996.565012.

[23] D. Steidl, B. Hummel, E. Juergens, Quality analysis of source code comments, in: ICPC 2013: Proceedings of the 21st IEEE International Conference on Program Comprehension, San Francisco, CA, USA, 2013, pp. 83–92.

[24] N. Khamis, R. Witte, J. Rilling, Automatic quality assessment of source code comments: the JavadocMiner, in: NLDB 2010: 15th International Conference on Natural Language & Information Systems, Cardiff, UK, 2010, pp. 68–79.

[25] H. Zhong, Z. Su, Detecting API documentation errors, in: OOPSLA 2013: Object-Oriented Programming Systems, Languages, and Applications, Indianapolis, IN, USA, 2013, pp. 803–816.

[26] M. J. Kusner, Y. Sun, N. I. Kolkin, K. Q. Weinberger, From word embeddings to document distances, in: ICML 2015: Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015, pp. 957–966.