RepliComment: Identifying Clones in Code Comments

Arianna Blasi[♣♠] · Alessandra Gorla[♠]

[♣]Università della Svizzera italiana (USI), Lugano, Switzerland

[♠]IMDEA Software Institute, Madrid, Spain

ABSTRACT

Code comments are the primary means to document implementation and ease program comprehension. Thus, their quality should be a primary concern to improve program maintenance. While a lot of effort has been dedicated to detect bad smell in code, little work focuses on comments. In this paper we start working in this direction by detecting clones in comments. Our initial investigation shows that even well known projects have several comment clones, and just as clones are bad smell in code, they may be for comments. A manual analysis of the clones we identified revealed several issues in real Java projects.

KEYWORDS

Code comments, Software quality, Clones, Bad smell

ACM Reference Format:
Arianna Blasi[♣♠] · Alessandra Gorla[♠]. 2018. RepliComment: Identifying Clones in Code Comments. In *Proceedings of ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , Gothenburg, Sweden, May 27–28, 2018 (ICPC '18),* 4 pages.
https://doi.org/10.1145/3196321.3196360

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 procedures and classes 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 [3, 12, 13, 15]. 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 aim to identify and report *comment clones.* 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 different issues: They may be instances of copy and paste errors, and therefore comments may not match their corresponding implementation. Otherwise, they may simply provide poor information, which may not be useful to understand the semantic of the implementation. Although these cases are not issues per se, our analysis shows that most of the times these clones point to documentation that could be improved.

Corazza et al. conducted a manual assessment on the coherence between the comments and the implementation, and found instances of comment clones [2]. Similarly, Arnaoudova et al. [1] found some comment clones in their study about Linguistic Antipatterns in software. 93% of interviewed developers considered such issues as a poor/very poor practice. These studies show that the comment clone problem exists and it is relevant for developers.

In this paper we present RepliComment, a technique that automatically identifies comment clones, which may be symptoms of issues that developers want to fix. We used RepliComment to analyze the code base of 10 well-established Java projects. Our preliminary evaluation highlights that even solid and well known projects contain comment clones, and several of them should be analyzed and fixed by developers to improve the quality of documentation.

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, our technique and corresponding prototype to identify comment clones. Section 4 presents the results of our preliminary evaluation. Section 5 discusses some related work, and Section 6 concludes and discusses the future research direction of this work.

2 COMMENT CLONES

We found that there exist very different types of comment clones. In this section we try to highlight the most common scenarios. The comment clones we are mostly interested in are the ones that are misleading and do not match the implementation they document.

One example of this type of clone is the following, which we found in the Google guava project in release 19:[1]

```
/**
 * @return true if this matcher matches every character in the
 * sequence, including when the sequence is empty.
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196360

[1]http://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/CharMatcher.html#matchesNoneOf(java.lang.CharSequence)

```
 */
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.*

In this example, 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 semantic of its name, while it does match the semantic of matchesAllOf(). This clone is an example of a copy and paste error. It is likely that 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 the filter they apply is different. In the first case the filter returns all the elements matching a given pattern, while in the second case it returns all the elements that do not match the given pattern. Copying comments is not intrinsically bad practice, unless developers forget to modify the pasted comment according to the implementation, as in this case. This is a type of problem we want to identify and report to developers, so they can fix it.

Comment clones may also be examples of poor information that could be improved to offer a better understanding for developers. See the following example in the 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 little informative, since any boolean method returns either true or false. A more useful documentation should state what the boolean value represents. Such clones are symptoms of documentation that could be improved, and thus we aim to report them as well.

Finally, comment clones may occur for legitimate reasons, as when two methods offer the same functionality. See the following example:

```
/**
 * 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.*

This example can be found in class SolrClient of Apache solr, release 7.1.0. 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 that their method descriptions are identical. The difference between these two methods, which lays in their parameter lists, is properly documented through custom @param tags.

3 DETECTING COMMENT CLONES

The aim of RepliComment is to find comment clones and report them to developers so they can provide a proper correction. Our preliminary implementation focuses on Javadoc comments. They may refer to classes or methods, and usually include a short free-text description followed by paragraphs semi-structured with tags. In case of methods, the tags can describe parameters, return values and exceptional behavior. Given their block structure, we believe they are more susceptible to copy-paste practice than inline comments.

Our research work started by studying several projects. The analysis identified the three main types of clones that we presented in Section 2. While we are interested in reporting the first two instances of clones, since developers should either fix them (Sample 1) or could provide better documentation (Sample 2), we do not want to report clones of the last instance, since they are legitimate. Beside overloading and overriding, which are obvious cases, there are many reasons why a comment clone may be legitimate – a parameter may be common to different methods, as well as thrown exceptions, and these cases correctly share the same documentation. With RepliComment we aim to exclude as many legitimate cases of comment clones as possible, since we want developers to focus on the critical comment clones. RepliComment takes the URL of a project's GIT repository, and reports comment clones in such project working along the following steps:

(1) It clones the source code of the project, and identifies source code files to analyze.
(2) For each source, RepliComment identifies the list of methods, and for each method it parses the Javadoc classifying each paragraph as:
   - free-text, which is the paragraph that may be present at the beginning of the block comment.
   - @param tags, which are the comment blocks that describe parameters.

- @return tag, which is a block of comment that describes the return value of the method.
- @throws tags, which are comment blocks describing possible exceptional behaviors.

RepliComment extracts and parses each Javadoc comment block by means of the JavaParser library.[2]

(3) RepliComment compares each separate block with the same type of comment of other methods of the same source file. So, it compares each @param tag comment with other @param comments and so on. RepliComment currently only works at the file granularity level, but in the future we plan to extend it to other granularity levels (e.g. package or project). However, we believe that most copy and paste errors for comments occur within the same file.

(4) When RepliComment finds that two or more blocks of Javadoc comment are clones, it runs automatic checks to decide whether the clone might be legitimate or not. RepliComment considers a clone as potentially legitimate, and thus does not report it, if it falls in one of the following cases: 1) the clones belong to methods with equal names (as it is the case for function overload and override), 2) the comment describes the same exception type, and 3) the clones affect parameters that have the same name. Notice that when developers clone an entire block of Javadoc (e.g. free-text together with all the tags), they will copy also parameters names. A trivial automated check would think that @param tags clone are legitimate, even if they refer to non-existing parameters for a method. We avoid this problem by checking the coherence between @param tags and the list of parameters of the documented method. When incoherent, RepliComment reports the clone as non-legitimate.

(5) Clones that are classified as potentially problematic, i.e. those that do not fall in the previously mentioned cases, are stored in a CSV file with the following information: 1) fully qualified name of the class; 2) signature of the first method; 3) signature of the second method; 4) type of cloned Javadoc comment (i.e., free-text, @param, @return or @throws); and 5) cloned text.

The CSV file that RepliComment generates must then be manually inspected, as it may contain other legitimate clones which should be ignored. In the next section we present a preliminary empirical evaluation of using RepliComment on real Java projects.

4 EARLY EMPIRICAL EVALUATION

For our empirical evaluation we selected and analyzed 10 projects among the most popular and largest repositories on GitHub. We used RepliComment to analyze the whole set of Java source files, and for the purpose of the study we keep track of any comment clone that it reported, whether legitimate or not.

For each project we manually analyzed every clone stored in the CSV. We mostly focused our attention on cases that RepliComment reports as non-legitimate, since they likely indicate an issue. We manually analyzed the clones that RepliComment report as “legitimate”, to assure the correctness of their classification. We could easily confirm some non-legitimate clones as potential issues by reading the method signatures. In most cases it was easy to understand which method was correctly associated to a comment and which one is the result of a copy-paste error. For other cases, instead, it was necessary to look at the source code. In rare cases, it was not possible to establish which comment was the right one and which one was the clone.

Table 1: List of analyzed projects with corresponding comment clones identified by RepliComment

Project	Legit	C&P	Poor info	Unclear	Total
elasticsearch-6.1.1	877	12	8	1	898
hadoop-common-2.6.5	297	7	19	0	312
vertx-core-3.5.0	5559	7	0	2	5568
spring-core-5.0.2	270	4	3	1	278
hadoop-hdfs-2.6.5	76	3	23	0	102
log4j-1.2.17	3757	2	0	0	3759
guava-19.0	267	1	0	0	268
rxjava-1.3.5	2301	0	0	0	2301
lucene-core-7.2.1	182	0	0	0	182
solr-7.1.0	539	0	0	0	539
Total	14141	36	53	4	

Table 1 shows the results of our manual examination of the automatically produced reports. The “Legit” column shows the amount of comment clones that turned out to be legitimate after our manual evaluation. The “C&P” column stands for "Copy and paste", and it contains the number of clones that are probably the results of a comment copied from one method and then just pasted to another one without further corrections. The “Poor Info” column contains the amount of comment clones that give generic information that could potentially document a large number of methods at the same time. Most of these cases represent documentation that could be improved with more specific information. A comment clone that we often found was “throws Exception on error”. Such a comment can describe any kind of documented exception. Still, it is not useful for a developer, since it does not describe the conditions that raise the exception. The last column reports the comment clones that we could not properly classify. We found only 4 of such cases.

Overall, we found 89 relevant issues across the 10 projects, plus 4 clones for which the classification was not sure. The 89 issues include 36 copy and paste errors, which we consider critical issues that should be fixed, and 53 instances of comments that could be improved. Copy and paste issues are present in 7 projects out of the 10 we analyzed, while the poor information issues are only in 4 out of 10. An interesting finding is that 79% of the 53 cases of poor information belong to a single project (Hadoop). It is reasonable to suppose that developers maintain the same documentation style within the same project, and probably Hadoop developers do not mind having generic information documenting their code. It is also interesting to notice that only 3 projects out of the 10 we analyzed do not contain any issue: rxjava, lucene-core and solr.

5 RELATED WORK

A lot of work on clone detection focuses on code [10]. Typically, code clone detection techniques remove comments, as well as whitespaces and tabs, from the source code to eliminate spurious

[2]https://github.com/javaparser/javaparser

information [4, 6, 10]. Indeed, considering comments while searching for code clones could lead to miss some relevant code clones that differ in their comment descriptions. The work by Marcus et al. is an exception to this practice [8]. 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 [9]. Nonetheless, the aim of our work is different from general code clone detection. The Javadoc clones that RepliComment report typically do not identify 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 work on this respect has been done. Steidl et al. have some purposes in common with our work [11]. They study techniques to assess the coherence between comment and code. They compare the lexical similarity of comment 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 such a technique. We believe this problem can be addressed more precisely, for example, via a more comprehensive semantic analysis. Khamis et al. developed JavadocMiner [5], a tool that assesses the overall quality of Javadoc comments. They measure the comment quality through classic NLP metrics (such as readability index). However their main purpose is to verify that the Javadoc standard is correctly used - e.g., @param tags comment should start with the name of the documented parameter. Another work about comment quality by Zhong et al. [14] 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 implemented RepliComment, a prototype to automate the identification and classification of source comment clones that may be worth 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 presence of copy-paste issues, for instance, we could automatically identify which method is the source, and thus which comment should be fixed by developers. We could employ natural language analysis on the cloned comment and their corresponding method signatures, and report the mismatching cases. There are different techniques in the state of the art to asses document similarities, such as Word Embedding [7]. We could compare the semantic of method names with the semantic of their corresponding comments. We would report as likely to fix the comment clone whose method name is less similar to the comment. The analysis for "poor information" clones could benefit from additional metrics. There exist different metrics to assess text characteristics, such as its complexity, its quality and the quantity of information it describes. The example reported in Section 4 "@throws exception on error" would likely be classified by these metrics as poor text, and thus not informative. 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 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 better description".

The code of RepliComment is open source and available at:

https://github.com/ariannab/replicomment

ACKNOWLEDGMENTS

This work was supported by the EU FP7-PEOPLE-COFUND project AMAROUT II (n. 291803), by the Spanish project DEDETIS, and by the Madrid Regional project N-Greens Software (n. S2013/ICE-2731).

REFERENCES

[1] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. Gueheneuc, and G. Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *ICSM 2010: 26th IEEE International Conference on Software Maintenance*, pages 1–5, 2010.
[2] A. Corazza, V. Maggio, and G. Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *SQJ*, pages 1–27, 2016.
[3] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, 2016.
[4] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7):654–670, 2002.
[5] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: the JavadocMiner. In *NLDB 2010: 15th International Conference on Natural Language & Information Systems*, pages 68–79. Springer, 2010.
[6] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE 2007: 14th Working Conference on Reverse Engineering*, pages 170–178, 2007.
[7] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. From word embeddings to document distances. In *ICML 2015: Proceedings of the 32nd International Conference on Machine Learning*, pages 957–966, 2015.
[8] A. Marcus and 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*, pages 107–114, 2011.
[9] J. Mayrand, C. Leblanc, and 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*, 1996.
[10] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, School of Computing, 2007.
[11] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *ICPC 2013: Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 83–92, 2013.
[12] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *ICST 2012: 5th International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.
[13] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. Automatic model generation from documentation for Java API functions. In *ICSE 2016: Proceedings of the 38th International Conference on Software Engineering*, pages 380–391, 2016.
[14] H. Zhong and Z. Su. Detecting api documentation errors. In *OOPSLA 2013, Object-Oriented Programming Systems, Languages, and Applications*, pages 803–816, 2013.
[15] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing APIs documentation and code to detect directive defects. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*, pages 27–37, 2017.