Empirical Analysis of Words in Comments Written for Java Methods

Hirohisa Aman*, Sousuke Amasaki†, Tomoyuki Yokogawa† and Minoru Kawahara*
*Center for Information Technology, Ehime Univ., Matsuyama, Ehime 790-8577, Japan
†Faculty of Computer Sc. and Systems Eng., Okayama Prefectural Univ., Soja, Okayama 719–1197, Japan

***Abstract*—This paper focuses on comments written in source programs. While comments can work for improving the readability of code, i.e., the quality of programs, there have also been concerns that comments can be added for complicated source code in order to compensate for a lack of readability. That is to say, well-written comments might be associated with problematic parts to be refactored. This paper collected Java methods (programs) from six popular open source products, and performs analyses on words which appear in their comments. Then, the paper shows that a method having a longer comments (more words) tends to be more change-prone and would be required more fixes after their releases.**

I. Introduction

Comments in source programs are widely known as embedded documents and useful artifacts related to the code quality. Programmers often write comments into their source programs. For example, comments comprise about $15 - 20\%$ of lines of Linux and Mozilla source code [1]. Developers and maintainers place importance on comments in the program comprehension [2]. In many cases, comments can help to improve readability and understandability of programs [3].

Although the usefulness of comments is well-known [2], [4], [5], there have also been concerns in regard to comments. In the code refactoring world, comments have been said to be associated with code smells [6]. While comments may help the program comprehension, they are sometimes written to mask code smells of complicated code fragments. That is to say, well-written comments may work as deodorants in smelling programs. Buse et al. [7] mentioned that there are possibly comments to compensate for a lack of readability in a hard-to-understand program. Aman et al. [8], [9] conducted empirical studies on comments and reported that commented programs are more likely to be fault-prone or change-prone than non-commented ones. However, those reports focused only on coarse-grained data, i.e., "the presence or absence of comments" or "lines of comments." In order to make more sophisticated discussions on comments, we will conduct a finer-grained analysis and report our findings in this paper.

The remainder of this paper is organized as follows: Section II explains comments of interest in this paper and briefly describes the related work along with our motivation. Then, the section presents our research question (RQ). Section III reports our empirical study analyzing six popular open source products and discusses the results based on our RQ. Finally, Section IV presents our conclusion and future work.

II. Comments and Related Work

A. Comments of Interest

According to Steidl et al. [10], comments are classified into following seven types: 1) Copyright comments; 2) Header comments; 3) Member comments; 4) Inline comments; 5) Section comments; 6) Code comments; 7) Task comments.

Type 1 comments give the copyright designation or the license information of the code; Type 2 comments present an overview of the class (module) including its functionality, author, version etc.; Type 3 comments are the descriptions on methods (functions) or fields (global variables); Type 4 comments give implementation decisions within a method body; Type 5 comments are separators for a set of methods or fields. Steidl et al. showed "// --- Getter and Setter Methods ---" as an example. Type 6 comments are disabled code (comment out); Type 7 comments present developers' notes such as TODOs, memos regarding bugs, etc.

In this paper, we analyze comments which are related to a method, in accordance with previous work [8], [9]. For each method, we can see two types of comments corresponding to their positions: a) comments which are written right before a method, and b) comments which are written inside a body of method. We will refer to a) and b) as "documentation comments" and "inner comments," respectively. Documentation comments include Type 3 comments written for methods. In Java, they can be used for a manual generation if they are written in Javadoc format. Inner comments include Type 4 and Type 7 comments; while Type 6 comments also seem to be included, we will exclude the type of comments from our inner comments because of their different purposes.

B. Related Work

There have been concerns in regard to comments in the past. Fowler [6] said that well-written comments may be related to code smells. Although comments may present an additional information for a better understanding of the program, they can also play a role of deodorant masking code smells. In such a case, it would be better to improve the clarity of code. Kernighan et al. [11] recommended to rewrite a code rather than adding comments when a programmer faces a bad code.

Steidl et al. [10] focused on long comments having over 30 words, and conducted a survey on those comments. According to their report, long comments tend to contain important information which we cannot get from only the source code.

In other words, those parts are especially complex and hard-to-understand without comments. Moreover, a presence of such long comments means a lack of adequate external documents. That is to say, there is possibly a technical debt in such a part.

Aman et al. [8], [9] analyzed comments appearing in open source products, and showed that methods having one or more inner comments are more likely to be modified after their releases than the others. Thus, inner comments can be clues to find potentially problematic methods which should be preferentially reviewed. However, their analyses missed the content of comments: a finer-grained analysis would be useful for more sophisticated applications of studies.

C. Research Question

Toward a further comment analysis, we tackle the following research question (RQ) with a data collection which is finer-grained than the previous work [8], [9]:

RQ: Do word-level features of comments associate with change-proneness of programs?

□

By checking words appearing in documentation comments and inner ones, we can see general trend of comments and differences among products. Since the previous work [8], [9] focused only on the existence of comments or the lines of comments, this study performs a more detailed analysis in order to understand the impacts of comments in more depth. The above RQ is a question for an application of our data. If we show a usefulness of word-level analysis for predicting change-prone methods, we will be able to enhance the comment-based quality evaluation and prediction studies along with the previous work.

III. Empirical Study

A. Aim and Dataset

The aim of this study is to examine comments written in source programs in accordance with the above RQ, and to report the empirical results and findings.

Our dataset consists of programs from six popular open source software (OSS) products: Elasticsearch[1], Fastjson[2], Guava[3], libGDX[4], Presto[5] and RxJava[6]. The main reasons why we selected them are as follows: 1) Their source programs are written in Java; 2) Their source programs are maintained with Git; 3) They are popular and not small-sized products.

The reason 1 is from the restriction of our data collection tools[7]: JavaMethodExtractor and CommentExtractor. The reason 2 is for ease of fine-grained code analysis: Git allows making a clone of repository on our local disk, and provides powerful functionality for tracing code changes. The reason 3 is for the generality of our results. Results derived from minor or small-sized products would not be attractive for many developers. All of the above six products are ranked in the top 30 Java products in terms of "stars" at GitHub, and their total sizes of source files are greater than 100 KLOC.

B. Data Collection

We conducted our data collection in the following procedure for each product.

1) For each Java source file, trace its change logs and detect when (which commit) it was upgraded.
2) For each upgraded version of each Java source file, extract all Java methods and their comments by using our tools JavaMethodExtractor and CommentExtractor[8].
3) For each upgraded version of each Java source file, get the difference with its previous upgraded version, and detect which methods were upgraded at the commit: in particular, for each upgrade (commit) of a Java source file, check out both the older version and the newer version, and extract the source code fragments corresponding to the method of interest. Then compare two source code of the method by using the diff utility. When there is a difference between them, we regard that the method was upgraded through the commit.

   Through the above three steps, for each Java method, we can obtain its change history and comments.

4) Perform a data preprocessing: extract tokens appearing in comments, and transform them to their base forms using TreeTagger[9]. Then, omit HTML or Javadoc tags such as "@param," symbol strings such as "---" and stop words[10] such as "at," "it," "the," etc. Finally, obtain the set of words from them. While there would be non-English words or abbreviated words which are from identifiers' names, we include these words into our set of words as well because of an ease of data processing; A more sophisticated data processing is our future work.
   Stop words are ones which commonly and frequently appear in documents, and they are often removed before a document analysis in order to capture the characteristics of documents more accurately [12]. Since HTML tags, Javadoc tags and symbols may also be stop words in the case of program comments, we decided to remove them before our comment analysis as well.

   We show a simple example of the above word extraction. Figure 1 presents a part of method which has inner comments. The content of comments can be analyzed by TreeTagger: Fig. 2 shows the results where each line corresponds to each token and the first, second and third elements of each line are the original form, the type of the word (part-of-speech tag) and the base form, respectively. For example, token "was" in the third line is a past tense

[1]https://github.com/elastic/elasticsearch
[2]https://github.com/alibaba/fastjson
[3]https://github.com/google/guava
[4]https://github.com/libgdx/libgdx
[5]https://github.com/prestodb/presto
[6]https://github.com/ReactiveX/RxJava
[7]http://se.cite.ehime-u.ac.jp/tool/

[8]CommentExtractor can extract comment lines from Java method's source code and guess their types of comments, including inner comments, documentation ones and comment-outed code.
[9]http://www.cis.uni-muenchen.de/%7Eschmid/tools/TreeTagger/
[10]http://www.textfixer.com/tutorials/common-english-words.txt

```
.....
if (index == null && shard == null && primary == null) {
  // If it was an empty body, use the "any unassigned shard" request
  return new ClusterAllocationExplainRequest();
} else if (index == null || shard == null || primary == null) {
.....
```

Fig. 1. An example of inner comments (from Elasticsearch).

If	IN	if
it	PP	it
was	VBD	be
an	DT	an
empty	JJ	empty
body	NN	body
,	,	,
use	VBP	use

the	DT	the
“	“	”
any	DT	any
unassigned	JJ	unassigned
shard	NN	shard
”	”	”
request	NN	request

Fig. 2. Analysis results of comments shown in Fig.1 by TreeTagger.

of verb (VBD) and its base form is “be.” Then, symbols—comma and double quotation—and stop words—if, it, be, an, the, any—are omitted. The remaining words are “empty,” “body,” “use,” “unassigned,” “shared” and “request.”

C. Analyses and Results

There are $118,989$ methods (including constructors; excluding abstract ones) in six products and $17,143$ ones have comments (see Table I). We analyze those commented methods.

In order to capture features of comments in terms of their words, we conducted two preliminary studies—studies on words appearing in comments and on amounts of comments.

1) Preliminary Study on Words appearing in Comments: Table II shows frequently appearing words (ranked in the top 10; in the decreasing order of appearance) of documentation comments for methods of initial version. The bold-faced and underlined words are the ones common to two or more products. From Table II, we can see that the majority of frequently appearing words are common across products. Most common words are related to the return values and types in regard to methods. Since documentation comments are usually programmer's manuals of corresponding methods, they are possibly written in a common style with common words.

Table III shows the similarities between sets of all words (not limited to the top 10) appearing in documentation comments: the similarity in this study is the Jaccard index, $J(W_i, W_j) = \mid W_i \cap W_j \mid / \mid W_i \cup W_j \mid$, where W_i and W_j are sets of words to be compared. The bold-faced and underlined values are greater than their average (0.264). From the table, we can observe that (b)Fastjson has only lower similarities. While Fastjson has many common words in terms of the top 10 words (see Table II), there seem to be many other words which appear only in Fastjson, and it may have a different trend in describing documentation comments.

Similarly, Tables IV, V show the results of inner comments. From Table IV, we can say that inner comments are written using words related to the return values and types as well as documentation comments. On the other hand, there seem to be different trends. It looks popular to describe points to be checked/confirmed because words “check” and “need’ appear as common words. Word “n’t” is part of “can’t” or “won’t,” so limitations or things what programmers should not do may be frequently described in inner comments.

Since Fastjson does not have any common words in the top 10 (see Table IV) and it also has only lower similarities than the average (0.267), Fastjson would have a different tendency of comments. While (f)RxJava also has lower similarities, their differences from the average are only 3–13%. Thus, RxJava would not so differ from others except for Fastjson.

Through the analysis of appearing words, we could find a project which has a different trend in commenting code from the others—Fastjson. We may have to analyze such a peculiar project separately in order to perform a more sophisticated change-prone method prediction. On the other hand, the remaining projects have similar sets of words in their comments. That is to say, there seems to be a common trend of words appearing in comments.

TABLE I
NUMBER OF METHODS IN THE SURVEYED OSS PRODUCTS.

Product	having no comment	having comments: only doc.	having comments: only inner	having comments: both	total
Elasticsearch	21,164	2,549	686	174	24,573
Fastjson	1,688	193	249	46	2,176
Guava	15,149	4,425	1,207	739	21,520
libGDX	31,508	2,886	895	222	35,511
Presto	20,106	259	1,052	36	21,453
RxJava	12,231	1,068	331	126	13,756
Total	101,846	11,380	4,420	1,343	118,989

TABLE II
FREQUENTLY APPEARING WORDS IN DOCUMENTATION COMMENTS.

product	words
(a) Elasticsearch	**use**, query, **return**, shard, name, **new**, **value**, request, field, **set**
(b) Fastjson	byte, **method**, **array**, **type**, **value**, **item**, **vector**, reference, string, object
(c) Guava	**value**, **return**, **method**, **element**, **use**, call, key, contain, map, **array**
(d) libGDX	**specified**, **return**, **new**, **array**, **set**, **value**, buffer, **use**, matrix, **vector**
(e) Presto	**array**, **specified**, big, table, **value**, **type**, index, position, **use**, **element**
(f) RxJava	source, emit, **item**, **value**, publisher, operator, function, default, **type**, particular

TABLE III
JACCARD INDEXES BETWEEN WORD SETS OF DOCUMENTATION COMMENTS ACROSS PRODUCTS.

product	(a)	(b)	(c)	(d)	(e)	(f)
(a)	—	0.168	**0.398**	**0.389**	**0.269**	**0.336**
(b)	0.168	—	0.112	0.166	0.256	0.202
(c)	**0.398**	0.112	—	**0.360**	0.210	**0.278**
(d)	**0.389**	0.166	**0.360**	—	0.247	**0.301**
(e)	**0.269**	0.256	0.210	0.247	—	**0.264**
(f)	**0.336**	0.202	**0.278**	**0.301**	**0.264**	—

TABLE IV
FREQUENTLY APPEARING WORDS IN INNER COMMENTS.

product	words
(a) Elasticsearch	**use**, **n't**, **need**, nothing, **value**, case, **set**, here, **check**, node
(b) Fastjson	byte, skip, char, count, end, trim, illegal, separator, cast, last
(c) Guava	**n't**, **value**, **use**, **call**, **return**, exception, **set**, element, **check**, **null**
(d) libGDX	method, stub, **key**, **use**, run, **check**, **return**, **n't**, first, line
(e) Presto	**value**, **use**, **null**, position, block, column, **new**, partition, add, **key**
(f) RxJava	**n't**, current, request, child, **need**, **new**, producer, **value**, state, **call**

TABLE V
JACCARD INDEXES BETWEEN WORD SETS OF INNER COMMENTS ACROSS PRODUCTS.

product	(a)	(b)	(c)	(d)	(e)	(f)
(a)	—	0.140	**0.397**	**0.377**	**0.414**	0.257
(b)	0.140	—	0.117	0.136	0.153	0.164
(c)	**0.397**	0.117	—	**0.375**	**0.363**	0.231
(d)	**0.377**	0.136	**0.375**	—	**0.367**	0.231
(e)	**0.414**	0.153	**0.363**	**0.367**	—	**0.269**
(f)	0.257	0.164	0.231	0.231	**0.269**	—

2) Preliminary Study on Amounts of Comments:

Next, we counted words appearing in comments for each project. Table VI shows the distributions of word count: columns labeled as 25%, 50% and 75% correspond to the 25 percentile, the 50 percentile (the median) and the 75 percentile, respectively. Notice that there are 0's in the table since symbol strings and stop words were excluded from our dataset by the data preprocessing (see step 4 of Section III-B). For example, the number of words in comment "// ---" is regarded as 0 because "---" is a symbol string and not an English word.

From Table VI, for both documentation comments and inner comments, the majority of comments consist of 10 or less words: their frequencies show right-skewed distributions. That is to say, programmers tend to write short sentences using 10 or less words as their comments.

3) Change-Prone Method Analysis (for our RQ):

In this paper, we have conducted two preliminary studies about word-level features of comments. As the results, there seems to be a commonality among sets of words appearing in comments (see Tables II–V), and the majority of comments are short sentences (see Table VI). Therefore, we can consider comments which have unusual words or which are long sentences (with many words) to be abnormal.

Because of a diversity in unusual words and their small-sized samples, we will focus on word counts in our analysis below; we would like to analyze the impacts of unusual words as our significant future work.

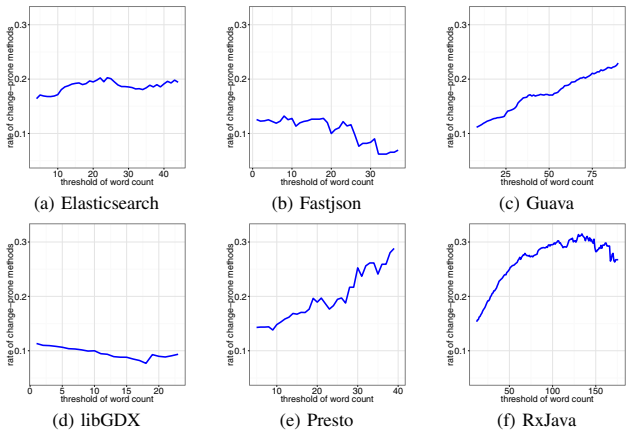
While the majority of comments are short sentences, there are so long ones consist of over 100 words (see Table VI). Such well-written comments may correspond to the concerns in the code refactoring world [6] as mentioned above. In order to examine the effect of word count on the code quality, we divide the set of methods into two subsets by using a threshold of word count, and compare them in terms of change-proneness. Now, we define "change-prone methods" to be the ones which has been frequently changed until the latest version and ranked in the top 10% of change counts. For a threshold value τ, we define $\mathrm{CR}_U(\tau)$ to be the rate of change-prone methods in the ones whose word count is greater than or equal to τ; we define $\mathrm{CR}_L(\tau)$ to be the rate of change-prone methods in the ones whose word count is less than τ. Figures 3, 4 show changes in $\mathrm{CR}_U(\tau)/\mathrm{CR}_L(\tau)$ over τ for documentation comments and inner comments, respectively.

For documentation comments (see Table 3), there seem to be mixed of increasing tendencies and decreasing tendencies. On the other hand, for inner comments, five out of six products (excluding (b)) show increasing tendencies (see Table 4). An increasing tendency means that a method having more words in its comments is more likely to be change-prone. That is to say, if a method has inner comments including many words, the method would require many times fixes after the release. The results partially prove the concern in regard to well-written comments which has been said in the code refactoring world. Since documentation comments play roles of manuals, those comments may not be associated with the above concern.

Although (b)Fastjson showed different tendencies, such

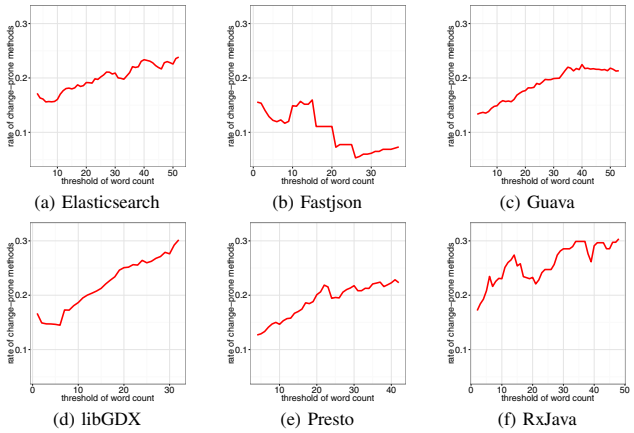
TABLE VI
DISTRIBUTION OF WORD COUNTS.

type	product	min	25%	50%	75%	max
doc	Elasticsearch	0	8	13	24	467
	Fastjson	0	0	5	22	205
	Guava	1	14	28	51	1,436
	libGDX	0	0	10	18	724
	Presto	1	9	13	25.5	110
	RxJava	0	21	42.5	116	492
inner	Elasticsearch	0	6	14	30	406
	Fastjson	0	0	2	10	129
	Guava	0	6	13	26	273
	libGDX	0	5	7	15	408
	Presto	1	6	13	23	342
	RxJava	1	2	5	16	800



(a) Elasticsearch (b) Fastjson (c) Guava
(d) libGDX (e) Presto (f) RxJava

Fig. 3. Rates of change-prone methods vs word counts in doc. comments.



(a) Elasticsearch (b) Fastjson (c) Guava

(d) libGDX (e) Presto (f) RxJava

Fig. 4. Rates of change-prone methods vs word counts in inner comments.

differences are possibly caused by low similarities of word set with the others. A further and detailed analysis would be required to explain the disparity.

D. Threats to Validity

In our data collection, we decided whether a method was upgraded at a commit or not, by checking the difference of the method's source lines before and after the commit. Since there have been a huge number of combinations of methods and commits in the repositories we surveyed, we were obliged to perform such a simple way of checking upgrades. Thus, there might be cases that only comments were upgraded through commits. Since such cases are minor ones in our datasets and we focused on the top 10% of change counts in our change-prone method analysis, we believe that the way of checking method upgrades would not be a serious threat to validity.

We excluded symbols from our dataset. If they had important meanings, the preprocessing affects our results. Since most excluded strings were separators like "---," our exclusion is not a serious threat in this study. For a more accurate analysis, we will need to build a corpus of comments in the future.

Comment descriptions can be affected by programmers' preferences. Thus, authors of comments might be influential on our results. Since our empirical study used only middle- or large-scaled products, threats caused by particular programmers are mitigated. An analysis of comment authors is one of our significant future work.

While we focused on words of comments, we miss the association between the code and comments. There might be misleading comments [1], [13] which confuse other developers or cause another fault injection. Moreover, some comments might be obsolete [14]. Such comments may have different kinds of impacts on the program comprehension and code changes in our dataset. A more sophisticated analysis which takes into account such associations between code and comments is our important future work.

IV. Conclusion and Future Work

We collected comments from six popular OSS products, and analyzed their trends of words and relationships with change-proneness of methods. Then, we got the following findings: (1) the majority of comments consist of short sentences with 10 or less words; (2) a method having more words in its inner comments is more likely to be change-prone.

Our empirical study partially proved the concern that well-written comments may be related to poor quality code, in terms of comment words. It is an easy way to detect a problematic method since we can find such a method by only checking comments, without complicated code metrics; thanks to editors such as Emacs, we can easily find comments since they are usually displayed in different styles or colors from executable code. Our findings can also be applied to the programming activity. When a programmer wants to add more comments (words) to his/her code, it would be better to perform a more careful review to the part and rewrite the code if needed.

Since the above trends may be affected by differences in programming domains, we plan to build comment corpora and perform a further analysis in the future. Moreover, we would like to perform further analyses focusing on (1) uncommon words appearing in comments and (2) misleading or obsolete comments as well in the future.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099. The authors would like to thank anonymous reviews for their helpful comments to an earlier version of this paper.

References

[1] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles*, Oct. 2007, pp. 145–158.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Int'l Conf. Design of Communication*, Sept. 2005, pp. 68–75.

[3] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sept. 1988.

[4] S. L. Pfleeger, *Software Engineering*. Upper Saddle River, NJ: Prentice Hall, 2001.

[5] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proc. Int'l Workshop on Mining Softw. Repositories*, May 2006, pp. 179–180.

[6] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman, 1999.

[7] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proc. Int'l Symp. Softw. Testing and Analysis*, July 2008, pp. 121–130.

[8] H. Aman, "An empirical analysis of the impact of comment statements on fault-proneness of small-size module," in *Proc. 19th Asia-Pacific Softw. Eng. Conf.*, Dec. 2012, pp. 362–367.

[9] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Lines of comments as a noteworthy metric for analyzing fault-proneness in methods," *IEICE Trans. Inf. & Syst.*, vol. E98-D, no. 12, pp. 2218–2228, Dec. 2015.

[10] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int'l Conf. Program Comprehension*, May 2013, pp. 83–92.

[11] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.

[12] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008.

[13] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston, MA: Prentice Hall, 2008.

[14] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Softw. Quality J.*, pp. 1–27, Nov. 2016.