Documentation Reuse: Hot or Not? An Empirical Study

Mohamed Oumaziz, Alan Charpentier, Jean-Rémy Falleri, Xavier Blanc

Documentation Reuse: Hot or Not? An Empirical Study

Mohamed A. Oumaziz, Alan Charpentier, Jean-Rémy Falleri, Xavier Blanc

CNRS, Bordeaux INP, Univ. Bordeaux
LaBRI, UMR 5800
F-33400, Talence, France
{moumaziz,acharpen,falleri,xblanc}@labri.fr

Abstract. Having available a high quality documentation is critical for software projects. This is why documentation tools such as Javadoc are so popular. As for code, documentation should be reused when possible to increase developer productivity and simplify maintenance. In this paper, we perform an empirical study of duplications in JavaDoc documentation on a corpus of seven famous Java APIs. Our results show that copy-pastes of JavaDoc documentation tags are abundant in our corpus. We also show that these copy-pastes are caused by four different kinds of relations in the underlying source code. In addition, we show that popular documentation tools do not provide any reuse mechanism to cope with these relations. Finally, we make a proposal for a simple but efficient automatic reuse mechanism.

Keywords: documentation, reuse, empirical study

1 Introduction

Code documentation is a crucial part of software development as it helps developers understand someone else's code without reading it [13, 25]. It is even more critical in the context of APIs, where the code is developed with the main intent to be used by other developers (the users of the API) that do not want to read it [12, 16, 17]. In this context, having a high quality reference documentation is critical [5].

Further, it has been shown that the documentation has to be close to its corresponding code [8, 9, 14]. Developers prefer to write the documentation directly in comments within the code files rather than in external artifacts [22]. Popular documentation tools, such as JavaDoc or Doxygen, all share the same principle which is to parse source code files to extract tags from documentation comments and to generate readable web pages [20, 24].

Writing documentation and code are highly coupled tasks. Ideally, developers should write and update the documentation together with the code. However, it has been shown that the documentation is rarely up-to-date with the code [8, 9, 14] and is perceived as very expensive to maintain [4, 5].

We think that one possible reason for this maintenance burden is that documentation tools lack reuse mechanisms whereas there are plenty of such mechanisms in programming languages. Developers that write documentation therefore copy-paste many documentation tags, which is suspected to increase the maintenance effort [11].

As an example, let us consider a case of delegation as shown in the Figure 1. In this example, the right method is just returning directly a value computed from the left method. As expected, some documentation tags from the left method are copy-pasted in the right method: the common parameters and the return value. As a consequence, if the documentation of the callee method is updated, an update of the caller documentation will have to be carried out manually, which is well known to be error-prone [11].

```
/**
* @param a the first collection, must
     not be null
* @param b the second collection, must
     not be null
* @return true iff the collections
     contain the same elements with the
     same cardinalities.
*/
public static boolean
     isEqualCollection(final
     Collection a, final
     Collection b) {
...
return true;
}
```

```
/**
* @param a the first collection, must
     not be null
* @param b the second collection, must
     not be null
* @param equator the Equator
     used for testing equality
* @return true iff the collections
     contain the same elements with the
     same cardinalities.
*/
public static  boolean
     isEqualCollection(final
     Collection a, final
     Collection b, final Equator
     equator) {
...
return
     isEqualCollection(collect(a,
     transformer), collect(b,
     transformer));
}
```

Fig. 1. Extract of a documentation duplication due to method delegation (in the Apache Commons Collections project). Duplicated tags are displayed in bold.

In this paper, we investigate this hypothesis and more formally answer the two following research questions:

- RQ1: Do developers often resort to copy-paste documentation tags?
- RQ2: What are the causes of documentation tags copy-paste and could they be avoided by a proper usage of documentation tools?

We answer our research questions by providing an empirical study performed on a corpus of seven popular Java APIs where the need of documentation is critical (see Section 2.1). We answer the first research question by showing how big is the phenomenon of documentation tags copy-pasting (see Section 3). To that extent, we automatically identify what we call documentation tags duplications (Section 2.2), count them, and manually check if they are intended copy-pastes

or just created by coincidence. We answer the second research question by investigating the intended copy-pastes we observed with the objective to find out their causes. Then we analyze whether existing documentation tools can cope with them (see Section 4). We further extend our second research question by providing a proposal for a simple but useful documentation reuse mechanism.

Our results show that copy-pastes of documentation tags are abundant in our corpus. We also show that these copy-pastes are caused by four kinds of relations that take place in the underlying source code. In addition, we show that popular documentation tools do not provide any reuse mechanism to cope with these relations.

The structure of this paper is as follows. First, Section 2 presents our corpus and the tool we create to automatically identify documentation duplications. Then, Section 3 and Section 4 respectively investigate our two research questions. Finally, Section 5 describes the related works about software documentation and Section 6 concludes and describes future work.

2 Experimental Setup

In this section, we first explain how we create our corpus (Section 2.1), and give general statistics about it. Then, we describe how we extract documentation duplications contained in our corpus (Section 2.2).

2.1 Corpus

The corpus of our study is composed of seven Java APIs that use JavaDoc, arbitrary selected from the top 30 most used Java libraries on GitHub as computed in a previous work of Teyton *et al.* [23]. We just considered the source code used to generate the documentation displayed on their websites. We also choose to focus only on methods' documentation, as this is where there is most of the documentation. In the remainder of this paper, we therefore only discuss about the documentation of Java methods written in JavaDoc.

Table 1 presents these seven APIs. All the data gathered for this study is available on our website[1]. As we can see in the *General* section of this table, the projects are medium to large sized (from 33 to 1,203 classes). As expected, they contain a fair amount of documentation: from about 28% to 97% of the methods are documented. The *Tags* section of this table gives some descriptive statistics of the JavaDoc tags used. As we can see, the most frequent tags are usually, in order: `@description`, `@param`, `@return` and `@throw`. Finally, the *inheritDoc* section of this table shows that there are few `@inheritDoc` tags. Such tags are used to express a documentation reuse between two methods but the method that reuses the documentation must override or implement the method that contains the reused documentation.

[1] http://se.labri.fr/a/ICSR17-oumaziz

Table 1. Statistics computed from our corpus and the documentation it contains.

	acc[1]	acio[2]	ggson[3]	Guava	JUnit	Mockito	SLF4J
General							
# of classes	466	119	72	1,203	205	375	33
# of methods	4,078	1,173	569	9,928	1,319	1,716	433
% of documented methods	61.53	97.27	52.55	36.37	43.44	28.15	36.49
Tags							
# of `@description`	1,939	922	265	3,073	448	436	128
# of `@param`	1,199	734	106	749	178	237	51
# of `@throw`	438	209	65	462	12	11	5
# of `@return`	892	322	92	414	90	131	42
inheritDoc							
# of usage	85	18	0	112	2	0	0

[1] Apache Commons Collections
[2] Apache Commons IO
[3] google-gson

2.2 Documentation Duplication Detector

A documentation duplication is a set of JavaDoc tags that are duplicated among a set of Java methods. If it is intended then it was created by a copy-paste, if not then it was created by coincidence. We propose a documentation duplication detector that inputs a set of Java source code files and outputs the so-called documentation duplications[2].

The detector first parses the Java files and identifies all the documentation tags they contain by using the GumTree tool [7]. To detect only meaningful duplications, it extracts the most important tags of JavaDoc: `@param`, `@return`, `@throws` (or its alias `@exception`). It also extracts the main description of Java methods as if it is tagged too (with an imaginary `@description` tag). Finally, to avoid missing duplications because of meaningless differences in the white-space layout, it cleans the text contained in the documentation tags by normalizing the white-spaces (replacing tabs by spaces, removing carriage returns and keeping only one space between two words). For the same reasons, it also transforms all text contained in documentation tags to lowercase.

As a next step, the detector makes a comparison between tags, and checks if they are shared between different Java methods. Table 2 shows the result of this step w.r.t. to the Java code of Figure 1.

The third and last step of the process consists in grouping the Java methods and the tags they share with the objective to identify maximal documentation duplications. This step is complex as it can lead to a combinatorial explosion, but fortunately, it can be solved efficiently using Formal Concept Analysis [10] (FCA). FCA is a branch of lattice theory that aims at automatically finding

[2] `https://github.com/docreuse/docreuse`

Table 2. The methods and their respective tags computed from Figure 1 source code (duplicated tags are depicted in bold).

	@param a	**@param b**	**@return**	@param equator
I1: isEqualCollection(final Collection a, final Collection b)	X	X	X	
I2: isEqualCollection(final Collection a, final Collection b, ...)	X	X	X	X

maximal groups of *objects* that share common *attributes*. In our context, the objects simply correspond to Java methods, and the attributes correspond to documentation tags.

FCA returns a hierarchy of so-called *formal concepts*. A formal concept is composed of two sets: the extent (a set of Java methods in our context) and the intent (a set of documentation tags in our context). The extent is composed of objects that all share the attributes of the intent. In other words in our context, a formal concept is a collection of Java methods that share several documentation tags.

The hierarchy returned by FCA then expresses inclusion relationships between the formal concepts. The Figure 2 shows such a hierarchy from the formal context of Table 2. To identify duplicated documentation tags, we search within the hierarchy the concepts that have at least two objects in their extent, and discard all other concepts as they do not correspond to duplications. The formal concepts corresponding to maximal duplications are shown in plain line in Figure 2, the others are not relevant in our context. In our example, one maximal documentation duplication has been identified.

3 Research Question 1

In this section, we answer our first research question: *Do developers often resort to copy-paste documentation tags?*. To investigate if documentation duplications are frequent, we simply apply our documentation duplication detector to our corpus and report statistics about the extracted duplications. To ensure that these duplications are intentional, we draw at random a subset of the extracted duplications and ask three developers to manually decide for each duplication if it is intentional or coincidental.

3.1 Frequency of Duplications

As shown in the *Documentation* part of Table 3, our detector has identified about $2,800$ documentation duplications in the seven APIs of our corpus. As we

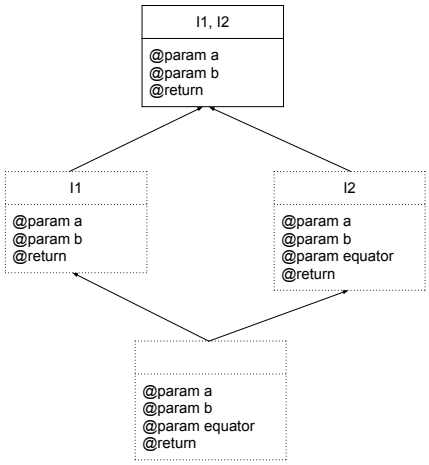

Fig. 2. Hierarchy computed by FCA from the formal context of Table 2. Retained concepts are depicted with plain lines, while discarded concepts are depicted with dotted lines.

can see, at most 4% of the documented methods have their documentation completely duplicated (line `% of complete dupl.`). This indicates that completely duplicating a method's documentation is rare. On the contrary, about 40% to 75% of the documented methods have their documentation partially duplicated (line `% of partial dupl.`). This indicates that duplicating some method's documentation tags is very frequent, at least much more frequent than using the `@inheritDoc` tags as seen in the Table 1, which raises questions about the limitations of this mechanism as we will see in Section 4.

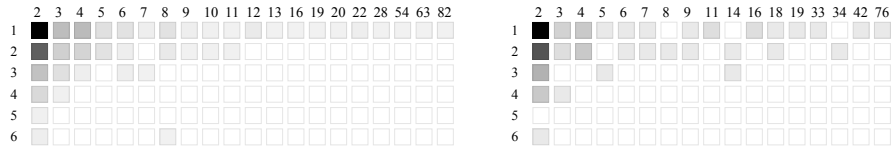

Fig. 3. Diagrams presenting the number of Java methods and tags of the identified duplications in Guava (left) and Mockito (right). The rows correspond to the number of Java methods, and the columns correspond to the number of documentation tags. The color in a cell correspond to the number of duplications (the darker the cell is, the more duplications).

In addition, we can see in the *Documentation tags* part of Table 3 a fine-grained analysis of the duplicated tags. Even though the frequency of duplications for each tag depends on the project, `param` and `throws` are often the most duplicated tags (from 20% to 40% of these tags are duplicated). The `return`

tags are also largely duplicated (from 14% to 31%). Finally, the `description` tag is rarely duplicated (from 4% to 15%).

Table 3. Statistics computed from our corpus and the duplications it contains.

	acc[1]	**acio**[2]	**ggson**[3]	**Guava**	**JUnit**	**Mockito**	**SLF4J**
Documentation							
# of dupl.	1,137	684	59	630	134	86	36
% of complete dupl.	1.51	4.03	0.67	0.80	0.00	0.00	0.00
% of partial dupl.	75.53	77.83	48.49	38.72	42.76	58.59	73.42
Documentation tags							
% of dupl. `@description`	11.24	14.53	8.30	6.57	14.96	4.13	3.91
% of dupl. `@param`	41.78	42.78	22.64	30.71	37.08	17.72	39.22
% of dupl. `@throw`	49.09	55.98	21.54	33.33	8.33	27.27	40.00
% of dupl. `@return`	27.35	31.37	15.22	25.85	14.44	19.85	26.19

[1] Apache Commons Collections
[2] Apache Commons IO
[3] google-gson

Figure 3 presents the number of documentation tags and Java methods of the identified duplications for Mockito and Guava projects. Due to the lack of space, we only show these two projects, but the figures are very similar for all the projects in our corpus, and can be found on our website. The Figure shows that most of the identified duplications have few documentation tags and few Java methods: many duplications involve only two Java methods that share a single documentation tag. On these figures, we notice that the maximum number of duplications' documentation tags ranges from 3 (in SLF4J) to 8 (in Apache Commons Collections). The maximum number of Java methods ranges from 22 (in google-gson) to 183 (in Apache Commons IO). Thus, there exist duplications involving a lot of method and only a few tags or a lot of tags and a few methods. Finally, there is no duplication with both a large number of methods and a large number of tags.

3.2 Copy-pastes vs Coincidental Duplications

To answer the second part of our research question we perform a qualitative experiment that relies on the manual judgement of several developers. We choose to involve three experienced Java developers for the experiment, as advised in [2], because judging if a duplication is an intended copy-paste or not is subjective. Involving three developers allows us to have a trust level on the status of a duplication. In our experiment, the developers are three of the paper's authors.

We then decided to create a sample composed of 100 duplications randomly drawn from our dataset of identified duplications, representing about 5% of the population of that dataset. Due to limitations in time we had to limit our manual

analysis to 100 duplications, we randomly selected them to have a representative ratio of the corpus. Each of the 100 duplications was then presented to each developer through a web interface that also presented the associated code. The developers then had as much time as they needed to judge whether the duplication was an intended copy-paste or not. Of course, the developers were not authorized to talk about the experiment until its completion.

A duplication labeled as "intended" is called from now on a *copy-paste* while a duplication labeled as "not intended" is called an coincidental duplication. When a developer is not able to decide whether the answer should be "intended" or "not intended", he must label the duplication as "not intended", to ensure that the number of copy-pastes that are found is a solid lower bound. We therefore define the two following trust levels. First, a copy-paste has a "majority" trust level when it has been labeled as "intended" by at least two participants. Last, a copy-paste has a "unanimity" trust level when it has been labeled as "intended" by the three participants.

Finally, we apply the bootstrapping statistical method [6] on our sample to compute a 95% confidence interval for the ratio of copy-pastes in our corpus. The bootstrapping method is particularly well-suited in our context since it makes no assumption about the underlying distribution of the values.

Before presenting our experiment results, it should be noted that the developers replied an identical answer on 69 out of 100 duplications. This indicates that the task of rating a duplication is not too subjective. Moreover, on these 69 cases, the developers agreed on a copy-paste 68 times, and on a coincidental duplication only one time[3]. It means that agreeing on a copy-paste is easy while agreeing on an coincidental duplication is difficult.

The main results of the experiment are presented in the Figure 4. About 85% to 96% of the duplications are copy-pastes when using the majority trust level. When using the stricter unanimity trust level, about 57% to 76% of the duplications are copy-pastes. In both cases, more than half of the duplications are copy-pastes.

3.3 Threats to Validity

Our experiment bears two main threats to validity. First, the developers are authors of the paper, therefore, it could bias their answer when judging the duplications. Even if they took extra care to be as impartial as possible, replicating the study would enforce its validity and that it is why all the experiment's data is available.[4] Second, the results obtained from this experiment cannot be generalizable to all APIs, because we used the duplications of only seven Java open-source APIs. Even if we only considered well known and mature open-source projects for the experiment, it would be better to replicate the study with other APIs wether open-source or not and in various programming languages.

[3] http://se.labri.fr/a/ICSR17-oumaziz/RandomExperiment
[4] http://se.labri.fr/a/ICSR17-oumaziz

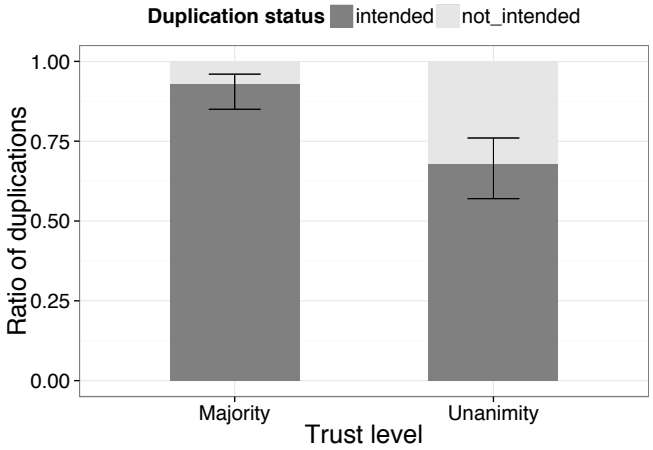

Fig. 4. Ratio of intended or coincidental duplications for the majority and unanimity trust levels, shown with their 95% confidence interval.

4 Research Question 2

In this section we answer our second research question: *What are the causes of documentation copy-pastes and could they be avoided by a proper usage of documentation tools?* We start by an analysis of the causes that lead to documentation copy-paste, and their root in the source code (see Section 4.1). Then we check whether the existing documentation reuse mechanisms can cope with these causes (see Section 4.2). Finally, we propose a new documentation reuse mechanism that can be used to cope with the unsupported causes (see Section 4.3).

4.1 Causes of Documentation Copy-pasting

To identify the causes that lead to copy-paste, we manually analyzed the source code corresponding to the 61 copy-pastes (68 initially with 7 containing only main tags which we did not keep) identified to answer RQ1 (see Section 3). After performing our manual analysis we identified four different causes: delegation, sub-typing, code clone and similar intent.

A *delegation*, as shown in Figure 1, appears when a method calls another one, and thus has a part of its documentation coming from the called one.

A *sub-typing* appears when a method overrides another one that is defined in a same hierarchy. In this case, it is common that the overriding method's documentation comes from the one of the overridden method.

A *code clone* appears when a method shares similar lines of code with another one, hence duplicating a part of its body. Figure 5 shows an example of code clone as the two methods share common lines of code.

```
/**
* Writes a String to the {@link
    StringBuilder}.
*
* @param value The value to write
*/
@Override
public void write(final String
    value) {
 if (value != null) {
  builder.append(value);
 }
}
```

```
/**
* Writes a portion of a character
    array to the {@link
    StringBuilder}.
*
* @param value The value to write
* @param offset The index of the
    first character
* @param length The number of
    characters to write
*/
@Override
public void write(final char[]
    value, final int offset, final
    int length) {
 if (value != null) {
  builder.append(value, offset,
      length);
 }
}
```

Fig. 5. Example of copy-paste due to code clone in the Apache Commons IO project. Duplicated tags are displayed in bold.

Finally, a *similar intent* appears when a method performs a computation that is similar to another method, which is why they share some documentation tags.

```
/**
* Delegates to {@link
    EndianUtils#
readSwappedInteger(InputStream)}
* @return the read long
* @throws IOException if an I/O error
    occurs
* @throws EOFException if an end of
    file is reached unexpectedly
*/
public int readInt() throws
    IOException, EOFException {
 return
 EndianUtils.readSwappedInteger(in);
}
```

```
/**
* Delegates to {@link
    EndianUtils#
readSwappedFloat(InputStream)}
* @return the read long
* @throws IOException if an I/O error
    occurs
* @throws EOFException if an end of
    file is reached unexpectedly
*/
public float readFloat()
    throws IOException,
    EOFException {
 return
 EndianUtils.readSwappedFloat(in);
}
```

Fig. 6. Extract of copy-paste due to two methods with a similar intent in the Guava project. Duplicated tags are displayed in bold.

Figure 6 shows such an example. Here the two methods only differ because of the return type (float or int). It is not a clone because there is no common line between them. Further, the funny thing is that the developer made a mistake as he clearly copied the documentation of the long method but didn't change the documentation of the int and float ones. In Java, most of similar intent cases we

observed are due to developers implementing several times a same feature for each primitive type.

Table 4 shows the occurrences of each relation in our corpus based on our analysis. We can see that the main cause of documentation copy-pastes is delegation (60%) and then code clone (28%). There are very few sub-typing (8%) and similar intent (3%) cases. Further, looking at the tag level we notice that this distribution is quite consistent whatever the tag.

Table 4. Percentage of copy/paste for each cause in our corpus.

	Cause			
	Delegation	**Sub-typing**	**Code clone**	**Similar intent**
copy-pastes	37/61(60%)	5/61(8%)	17/61(28%)	2/61(3%)
@description	8/15(53%)	1/15(7%)	6/15(40%)	0/15(0%)
@param	27/41(66%)	4/41(10%)	9/41(22%)	1/41(2%)
@return	18/30(60%)	3/30(10%)	8/30(27%)	1/30(3%)
@throw	18/30(60%)	1/30(3%)	9/30(30%)	2/30(7%)

4.2 Existing Documentation Tools

As a second step, we first look at the different documentation tools to obtain the mechanisms they provide for reusing documentation. As there are too many documentation tools (about fifty) [5], and due to time constraints, we choose to focus on the most popular ones. As a proxy to compute the popularity, we compute for each tool the number of questions asked by developers on StackOverflow, for the tools where a dedicated StackOverflow tag is available.

Table 5. The five documentation tools with the most questions in StackOverflow

Tool	**Language**	**#Questions**
JavaDoc	Java	2,022
Doxygen	C, C++, Java, C#, VBScript, IDL Fortran, PHP, TCL	1,894
phpDocumentor	PHP	636
JSDoc	JavaScript	574
Doc++	C, C++, IDL, Java	570

We then analyze in detail the five tools having the most related questions on StackOverflow, whether they are compatible with Java or not, in order to be sure

[5] `https://en.wikipedia.org/wiki/Comparison_of_documentation_generators`

that there is no mechanism available for other languages that could avoid copy-pastes and therefore should be implemented for Java. These tools are shown in Table 5.

As a second step for our experiment, for each tool, we go through the whole user-guide to find out the list of reuse mechanisms. We find out that these reuse mechanisms have two main aspects. The first aspect is about the reuse granularity: some allow only to reuse a whole method documentation, some allow to reuse documentation tags separately. The second aspect is about the location of the reused documentation. Some mechanisms only allow to reuse documentation located in an overridden method, some allow to reuse documentation located in any method. Table 6 summarizes the aspects of the mechanisms offered by the documentation tools.

Table 6. Aspects of the reuse mechanisms. Reuse granularity indicates if the documentation has to be completely reused (Whole) or if it is possible to select some tags (Choice). Source location indicates where can be the source of the reused documentation: in an overridden method (Override) or in a method anywhere in the code (Anywhere).

	Reuse granularity	
Source location	**Whole**	**Choice**
Override	JavaDoc, JSDoc	JavaDoc
Anywhere	Doxygen, JSDoc	

First, it is important to notice that only three tools out of five provide reuse mechanisms: DOC++ and phpDocumentor have no support at all to reuse documentation. More surprisingly, no tool supports the reuse of documentation tags anywhere in the code. Indeed, JavaDoc allows to reuse documentation tags, but only in an overridden method, while JSDOC and Doxygen allow to reuse complete method documentations in the code, but not specific documentation tags.

As a result for our classification, *delegation*, *code clone* and *similar intent* relations are not yet handled by any existing mechanism. On the contrary, duplications due to *sub-typing* relations are already properly handled by JavaDoc.

4.3 Documentation Reuse Revisited

Based on our findings, we suggest a novel mechanism to allow developers to automatically reuse documentation tags from a method to another one. Our proposal is an inline tag for JavaDoc. An inline tag can be used inside another tag, giving therefore the possibility to reuse the content of a specific tag but also to add more content before and after the reused one. We define it as: {`@reuse Class:Method(type[, type])[:TagName]`}.

For instance, by using this new mechanism, the documentation of the right method in Figure 1 becomes as in Figure 7.

```
/**
* @param a {@reuse Class:isEqualCollection(Collection, Collection)}
* @param b {@reuse Class:isEqualCollection(Collection, Collection)}
* @param equator the Equator used for testing equality
* @return {@reuse Class:isEqualCollection(Collection, Collection)}
*/
```

Fig. 7. Example of a documentation reuse with our `@reuse` inline tag.

As you can see in Figure 7, while using the `@reuse` tag inside `@param`, there is no need to specify which tag name to reuse, by default it will automatically reuse the tag with the same name as the tag it belongs to, therefore the `@param` named `a`.

We implemented our proposal as a doclet for JavaDoc, the source code can be accessed in our website[6]. By using this mechanism, it is possible to avoid at least all the copy-pastes due to *delegation*, the most frequent ones in our corpus (60% of the copy-pastes). While this mechanism could also be used for copy-pastes due to *code clone* and *similar intent*, one main problem would be to decide which method should be the documentation origin which is a still open research question. Finally, our proposal is not able to cope with `@description` tags, as they are not materialized in JavaDoc. We also plan to conduct a more thorough study of our proposal as a future work.

5 Related Work

This section describes the work done on the subject of software documentation: studies and tools. We start by describing the existing studies on this subject, which all agree on the fact that developers need more assistance for maintaining the documentation.

Forward et al. [9] perform a qualitative study on 48 developers and managers about how they feel about software documentation as well as the tools that support it. They discover that their favourite tools are word processors and Javadoc-like tools. They also discover that the participants think that the documentation is usually outdated. Finally, they discover that the participants would greatly appreciate tools that help in maintaining the documentation.

Dagenais and Robillard [5] perform a qualitative study involving 12 core open-source contributors writing documentation and 10 documentation readers. They analyze the evolution of 19 documentation documents across 1500 revisions. They identify three documentation production modes: initial effort, incremental changes and bursts (big amount of change in a small period). They also discover that Javadoc-like documentation is perceived as a competitive advantage for libraries, and is easy to create but costly to maintain.

Finally, Correia et al.[4] show that maintaining a documentation is highly challenging. They identify four so-called patterns to help tackling this challenge:

[6] http://se.labri.fr/a/ICSR17-oumaziz

information proximity, co-evolution, domain structured information and integrated environment.

In order to help creating and maintaining the documentation, several tools have been developed. We describe these tools in the remainder of this section, even if none of them supports documentation reuse as presented in our study.

DocRef [26] helps detecting errors in software documentation by combining code analysis and natural language processing techniques. This tool has then been validated on 1000 detected documentation errors from open-source projects, and has proven usefulness as many errors have been fixed after having been reported.

Childs and Sametinger [3] suggest the use of object-oriented programming techniques such as inheritance and information hiding in documentation to avoid redundancy. They also describe documentation reuse concepts and how to apply them using literate programming on documentation that is either or not related to source code.

Parnas [18] explains the lack of interest of researchers about the documentation topic. He further explains that his team and him developed a new mathematical notation that is more adapted for documentation but didn't convince academics and practitioners.

Buse and Weimer [1] present a tool that can statically infer and characterize exception-causing conditions in Java and then output a human-readable documentation of the exceptions. The tool is evaluated on over 900 instances of exception documentation within 2 million lines of code. They find out that the output is as good as or better than the existing one in the majority of the cases.

Pierce and Tilley [19] suggest using reverse engineering techniques to automate the documentation process. They propose an approach based on this principle in their Rational Rose tool. This approach offers the possibility to automatically generate up-to-date documentation. However their approach is not subjected to a serious evaluation.

McBurney and McMilla [15] describe a new method that uses natural processing language with method invocation analysis to generate a documentation not only explaining what the method does but also what is its purpose in the whole software project.

Robillard and Chhetri [21] describe a tool, Krec, that is able to extract relevant fragments of documentation that correspond to a given API element. The tool has been evaluated on a corpus of 1000 documentation units drawn from 10 open source projects and has shown to have a 90% precision and 69% recall.

6 Conclusion

Code documentation is a crucial part of software development. Like it is the case with source code, developers should reuse documentation as much as possible to simplify its maintenance.

By performing an empirical study on a corpus of seven popular Java APIs, we show that copy-pastes of documentation tags are unfortunately too abundant. By

analyzing these copy-pastes, we identified that they are caused by four different kinds of relationships in the underlying source code.

Our study pinpoints the fact that popular documentation tools do not provide any reuse mechanism to cope with these causes. For instance, there is definitely no mechanism supporting documentation reuse in the case of delegation, which is the major cause of copy-paste.

We looked towards a proposal providing a simple tag that makes the documentation reuse simple but efficient. As a further work, we obviously plan to extend our study. We plan to analyze other programming languages and documentation tools, and to detect not only identical documentations but also similar ones, aiming to find duplications with tiny differences. We finally plan to extend and validate our proposal from a developer point of view.

References

1. Buse, R.P., Weimer, W.R.: Automatic documentation inference for exceptions. In: Proceedings of the 2008 international symposium on Software testing and analysis. pp. 273–282. ACM (2008)
2. Charpentier, A., Falleri, J.R., Lo, D., Réveillère, L.: An Empirical Assessment of Bellon's Clone Benchmark. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering. pp. 20:1–20:10. EASE '15, ACM, Nanjing, China (2015), bibtex: charpentier_empirical_2015
3. Childs, B., Sametinger, J.: Literate programming and documentation reuse. In: Software Reuse, 1996., Proceedings Fourth International Conference on. pp. 205–214. IEEE (1996)
4. Correia, F.F., Aguiar, A., Ferreira, H.S., Flores, N.: Patterns for consistent software documentation. In: Proceedings of the 16th Conference on Pattern Languages of Programs. p. 12. ACM (2009)
5. Dagenais, B., Robillard, M.P.: Creating and evolving developer documentation: understanding the decisions of open source contributors. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 127–136. ACM (2010)
6. Efron, B., Tibshirani, R.J.: An Introduction to the Bootstrap. Chapman & Hall, New York (1993)
7. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and Accurate Source Code Differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 313–324. ASE '14, ACM, New York, NY, USA (2014)
8. Fluri, B., Würsch, M., Gall, H.C.: Do code and comments co-evolve? on the relation between source code and comment changes. In: Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on. pp. 70–79. IEEE (2007)
9. Forward, A., Lethbridge, T.C.: The Relevance of Software Documentation, Tools and Technologies: A Survey. In: Proceedings of the 2002 ACM Symposium on Document Engineering. pp. 26–33. DocEng '02, ACM, New York, NY, USA (2002)
10. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (1997)
11. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009. pp. 485–495 (May 2009)

12. Kramer, D.: API documentation from source code comments: a case study of Javadoc. In: Proceedings of the 17th annual international conference on Computer documentation. pp. 147–153. ACM (1999)
13. Lakhotia, A.: Understanding Someone else's Code: Analysis of Experiences. J. Syst. Softw. 23(3), 269–275 (Dec 1993)
14. Lethbridge, T.C., Singer, J., Forward, A.: How Software Engineers Use Documentation: The State of the Practice. IEEE Softw. 20(6), 35–39 (Nov 2003)
15. McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 279–290. ACM (2014)
16. Monperrus, M., Eichberg, M., Tekes, E., Mezini, M.: What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation. Empirical Software Engineering 17(6), 703–737 (2012)
17. Parnas, D.L.: A Technique for Software Module Specification with Examples. Commun. ACM 15(5), 330–336 (1972)
18. Parnas, D.L.: Software aging. In: Proceedings of the 16th international conference on Software engineering. pp. 279–287. IEEE Computer Society Press (1994)
19. Pierce, R., Tilley, S.: Automatically connecting documentation to code with rose. In: Proceedings of the 20th annual international conference on Computer documentation. pp. 157–163. ACM (2002)
20. Pollack, M.: Code generation using javadoc. JavaWorld, http://www. javaworld. com/javaworld/jw-08-2000/jw-0818-javadoc. html (2000)
21. Robillard, M.P., Chhetri, Y.B.: Recommending reference API documentation. Empirical Software Engineering 20(6), 1558–1586 (2015)
22. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A Study of the Documentation Essential to Software Maintenance. In: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information. pp. 68–75. SIGDOC '05, ACM, New York, NY, USA (2005)
23. Teyton, C., Falleri, J.R., Palyart, M., Blanc, X.: A study of library migrations in Java. Journal of Software: Evolution and Process 26(11), 1030–1052 (Nov 2014)
24. Van Heesch, D.: Doxygen (2004)
25. Vanter, M.L.V.D.: The documentary structure of source code. Information and Software Technology 44(13), 767 – 782 (2002), special Issue on Source Code Analysis and Manipulation (SCAM)
26. Zhong, H., Su, Z.: Detecting API documentation errors. In: ACM SIGPLAN Notices. vol. 48, pp. 803–816. ACM (2013)