Preliminary Report on Empirical Study of Repeated Fragments in Internal Documentation

Milan Nosál*, Jaroslav Porubán
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice
Letná 9, 042 00, Košice, Slovakia
Email: milan.nosal@gmail.com, jaroslav.poruban@tuke.sk

Abstract—In this paper we present preliminary results of an empirical study, in which we used copy/paste detection (PMD CPD implementation) to search for repeating documentation fragments. The study was performed on 5 open source projects, including Java 8 SDK sources. The study shows that there are many occurrences of copy-pasting documentation fragments in the internal documentation, e.g., copy-pasted method parameter description. Besides these, many of the copy-pasted fragments express some domain or design concern, e.g., that the method is obsolete and deprecated. Therefore the study indicates that the cross-cutting concerns are present in the internal documentation in form of documentation phrases.

I. INTRODUCTION

Preserving and comprehending developer’s concerns (intents) in the source code is still a current challenge in software development [1], [2], [3], [4]. In this paper we analyze the internal documentation (source code comments, JavaDoc, etc.) to recognize repeating documentation fragments that document those concerns (or features [5]). Our research question for this work is: Does internal documentation contain significant duplication? To answer this question we performed a copy/paste detection study, in which we analyzed JavaDoc comments in 5 open source projects. In this report we present preliminary results that indicate that there is a significant duplication of text in internal documentation. These repeating documentation fragments constitute documentation phrases discussed in several works – e.g., our previous work [6], or the one by Horie et al. [7]. This way the study has a potential to highlight the importance of those works and it may stimulate further attention to this topic.

II. DOCUMENTATION PHRASES

A documentation phrase is a set of documentation fragments with the same or similar formulation (part of sentence, sentence, a set of sentences) that can be found across the software system or even across multiple systems. Documentation fragments that represent the same documentation phrase usually document the same domain or design property that is shared by the documented program elements [6]. Horie et al. [7] likened the documentation phrases to crosscutting concerns from aspect oriented programming (AOP [8]).

As an example we can use the Swing library for component graphical user interfaces in Java. The library is not thread safe and therefore the programmer has to pay extra caution when using it in multithreaded systems (she has to use Event Dispatch Thread to safely work with the Swing components). Swing JavaDoc documents it and for each affected class includes a warning (see JPanel documentation in Figure 1).

Fig. 1. A NotThreadSafe documentation phrase instance in the Swing JPanel documentation

For this warning to be included in the JavaDoc, its HTML snippet has to be copy-pasted in the JavaDoc comment of each affected class.

III. DOCUMENTATION COPY/PASTE STUDY

In this study we analyzed the internal documentation of several Java frameworks and libraries to detect currently existing documentation phrases. In order to find documentation phrases, we performed a copy/paste detection1 on the documentation.

We have modified the PMD Copy/Paste Detection (CPD) tool2 to support copy/paste detection on JavaDoc documentation. The tool was fed preprocessed sources of several libraries and open source projects in Java and it analysed them to detect duplications in documentation that would indicate the potential

1Copy/paste detection is usually used to search for code that needs to be refactored, or to detect plagiarism [9].

2http://pmd.sourceforge.net/
Algorithm 1 JavaDoc preprocessing example – before

```java
package org.tuke;
/**
 * Dummy class.
 * Created by Milan on 5.3.2016.
 */
public class DummyClass { ...
```

Algorithm 2 JavaDoc preprocessing example – after

```java
public class DummyClass {
...
```

documentation phrases. We will discuss the process phases in more details in following sections.

A. Java Sources Preprocessing

In our experiment we used the tool to detect simple non-parametrized documentation phrases in the JavaDoc documentation. However, the PMD CPD was designed to be a code analysis tool and as such its purpose was to detect duplication in programming languages. Documentation phrases are fragments of documentation in a natural language.

PMD CPD tool works with language lexical tokens that it compares to detect duplications in their usage. To reduce tokenization complexity we pre-processed the sources to remove all the characters and tokens that are not the documentation. In other words the preprocessed sources are source files with only JavaDoc comments in their comments. Let us consider a simple class with JavaDoc from Listing 1.

This source file would be transformed to the content presented in Listing 2

Java lexical tokens were discarded along with asterisks indicating that following lines are part of JavaDoc (lines are preserved for backtracking to original sources). At the end of each JavaDoc comment we added a randomly generated unique "anchor" (DummyClass.jdoc.6.1394135902525.0.-2042003928. in the example) that prevented detection of duplicate chunks spanning multiple comments.

B. PMD CPD Modification

PMD CPD tool uses a tokenizer to read files and obtain lexical tokens of the language. In our experiment we used our custom tokenizer that divided the preprocessed files into sentences. Each sentence in the file was a single token. If we consider the DummyClass example from section III-A, the tokenizer would return following three tokens:

- "Dummy class."
- "Created by Milan on 5.3.2016."
- "DummyClass.jdoc.6.1394135902525.0.-2042003928."

Separators for the tokenization were characters ',' ', '?' and '!' followed by a whitespace character (therefore the date in the second token from the example was not divided in multiple tokens), or a new line character followed by an empty line.

C. Document Phrases Detection

For the duplication detection process we used the standard PMD CPD implementation (according to http://pmd.sourceforge.net/pmd-4.3.0/cpd.html they use Karp-Rabin string matching algorithm). We registered our modification in LanguageFactory and GUI classes and used the graphical user interface provided by the GUI class to run the tool.

In the setup of the copy/paste detection we set the 'Report duplicate chunks larger than:' option to a single token. This way PMD CPD reported even duplication of a single token – a single line in the documentation. The results were serialized as XML so that we could use XPath with XSLT to process them. First post-processing removed all the results that did not have at least 4 duplications – we considered 4 instances of a documentation phrase a reasonable threshold for considering it a significant documentation phrase.

IV. RESULTS

We performed the experiment on the following open source Java projects:

- sources of Java 8 standard edition4 with 7703 source files,
- PicoContainer5 with 1067 source files,
- JasperReports library6 with 2834 source files,
- JoSQL7 (SQL for Java Objects) with 85 source files, and
- jEdit8 with 573 source files.

A. Copy/paste Detection Results

The PMD CPD tool discovered 6102 duplicated fragments of various lengths in Java 8 source code. 2221 of them were duplicated fragments that had 4 instances. The highest number of instances of a single duplicated fragment was 344. Second highest were two duplicated fragments both with 299 instances. Figure 2 shows an overview of obtained results. We will provide a more detailed analysis of these data below.

PicoContainer contained 70 duplicated fragments ranging from 36 duplicated fragments with 4 instances to a single fragment duplicated 634 times. Figure 2 presents results for PicoContainer in a simple chart. Closer inspection of the results showed that duplicated fragments with the highest

---

3We used underscores here to highlight empty lines.


5https://github.com/picocontainer/picocontainer, commit 0f8172b

6http://sourceforge.net/projects/jasperreports/files/jasperreports/, version 6.0.0

7http://sourceforge.net/projects/josql/files/josql/, version 2.2

8http://sourceforge.net/projects/jedit/files/jedit/, version 5.2.0
In JasperReports the PMD CPD discovered 131 duplicated fragments. The most 'potent' duplicated fragment with 106 instances was a documentation phrase: '@deprecated To be removed.'. Parameter descriptions, especially this simple, could hardly be considered reasonable documentation phrases. The rest of results can be seen in Figure 5.

In JoSQL the PMD CPD discovered 31 duplicated fragments. The most 'potent' duplicated fragment with 54 instances was a parameter description: '@param q The Query object.'. Parameter descriptions, especially this simple, could hardly be considered reasonable documentation phrases. The rest of results can be seen in Figure 5.

jEdit project sources manifested 76 duplicated fragments with 4 or more instances. Again, the most 'potent' duplicated fragment (with 78 instances) was a line of asterisks. However, the second most 'potent' duplicated fragment was a sentence reporting that the documented method is thread-safe: 'This method is thread-safe.', thus showing that the thread-safe documentation phrase would be useful even beyond the scope of standard Java sources. The results overview can be seen in Figure 6.

We can conclude that duplicated fragments (documentation phrases) are common in practice.

V. Threats to Validity

We should mention several threats to validity that should to be considered for this study. First, the copy/paste detection found all the duplicated sentences in the documentation, even those that could hardly be assigned a concern. In those
cases using documentation weaving [7] would be impractical. Further examination would be useful.

Second, the modified PMD CPD detected solely static phrases – fragments of the documentation that were copy/pasted in documentation. Inclusion of parametrized documentation phrases [6] would be welcome.

VI. RELATED WORK

Maalej et al. in [10] present a study of knowledge patterns in API reference documentation. They define patterns as knowledge types that categorize types of information expressed by a particular documentation unit (a fragment of API documentation documenting one API element). As example we can mention types like the Functionality and Behavior knowledge type that describes what the API does or the Code Examples that provides a code sample showing how to use the API.

The work of Horie et al. [7] discusses documentation phrases from the aspect-oriented viewpoint. They view documentation phrases as cross-cutting concerns. Their tool ComponentWeaver is able to weave documentation phrases the same way as advices are woven in aspect-oriented programming. Our work presented in [6] continues in their work. There we propose using source code annotations to indicate program elements that should be documented by a given documentation phrase.

Shi et al. in [11] present an empirical quantitative study of API documentation evolution. They analyze the documentation to detect which parts of documentation are frequently revised, how often these revisions indicate behavioral changes in API and how often do these revisions occur. The contribution of their work is in emphasizing the importance of API documentation evolution in order to prevent defects in software using the given API.

VII. CONCLUSION

In conclusion, the presented results underline the significance of approaches like the one we presented in [6], or the one by Horie et al. [7], which centralize the management of such a documentation phrase into one place and thus ease their maintenance and evolution.

In the future work we need to further examine the results and confirm the significance of fragments that can be considered a concern (intent). An interesting modification of the experiment would be inclusion of the parametrized fragments as well.

ACKNOWLEDGMENT

This work was supported by project KEGA No. 019TUKE-4/2014 Integration of the Basic Theories of Software Engineering into Courses for Informatics Master Study Programmes at Technical Universities – Proposal and Implementation.

REFERENCES


