Javadoc Checker

Célia Cacciatore
celia.cacciatore@gmail.com
M2 IAGL

Jérémy Bossut
jeremy.bossut@gmail.com
M2 IAGL

Jonathan Geoffroy
geoffroy.jonathan@gmail.com
M2 IAGL

February 2015
# Contents

1 Introduction 3

2 Background 4
   2.1 Git ......................................................... 4
   2.2 Spoon ....................................................... 4

3 Requirements 4
   3.1 Mistakes in documentation .............................. 4
   3.2 Outdated documentation ................................. 5

4 Results 6
   4.1 Finding errors ........................................... 6
   4.2 Tests on Github projects ............................... 7
   4.3 Analysis on error types ................................. 8
   4.4 Limitations and Perspectives ......................... 10

5 Conclusion 11
1 Introduction

As a developer, it is always important to have reliable documentation to guide your work. If you are like most developers, you can be supporting multiple applications at the same time, which means that it is even more crucial to have documentation in place to help track all aspects of each application. It is also helpful for development, maintenance, and knowledge transfer to other developers.

“Incorrect documentation is often worse than no documentation.” [1]

However, the presence of documentation can be a double-edged sword when outdated. For instance, if you rely on faulty documentation, the code may be written perfectly properly to what the documentation says, but the bug truly lies in the documentation and not the code.

Besides, maintaining documentation over time is not a trivial process. First of all, documenting code is often seen as tedious and time consuming, especially because each change of code may imply a change in documentation as well. Moreover, a software project can have different kind of documentation, including not only source code documentation, but also user and developer documentation to mention just a few examples.

Our goal is to provide an automatic Javadoc Checker which analyzes the code using Spoon and informs the developer of bad practices in his documentation. In a git project, our application is also able to compare the current version with any of the previous ones in order to check if the documentation is still up-to-date.
2 Background

2.1 Git

Git[2] is a distributed version control system. We test our Javadoc Checker on git projects as it enables us to access different versions of the project and find changes in the code. Moreover, there are a lot of open-source projects on Github on which we can try our application.

2.2 Spoon

Spoon[7] is an Inria tool which enables to analyze and transform Java source code. It provides a new complete Java metamodel, including each element (classes, interfaces, methods...). It is far easier to understand than the Java Compiler AST.

3 Requirements

We define the bad practices our Javadoc Checker must be able to find in the code documentation. We distinguish between the mistakes that can be found directly in the project and the lack of changes that can be found by comparing two versions of the project.

3.1 Mistakes in documentation

We call mistake any imprecision or error we can find in the Javadoc. As it can have been generated by IDEs, it may become outdated if the methods change. We check that the documentation really corresponds to the portion of code it describes.

We list below the mistakes Javadoc Checker is able to find.
<table>
<thead>
<tr>
<th>Mistake</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing Javadoc for a class</td>
<td><code>public class HotelModel</code></td>
</tr>
<tr>
<td>Missing Javadoc for a method</td>
<td><code>public Hotel findHotelByID(int id)</code></td>
</tr>
<tr>
<td>Missing parameter in method description</td>
<td><code>/** * Returns an hotel with specific id * @return the specified hotel */</code></td>
</tr>
<tr>
<td>Unused parameter in method description</td>
<td><code>/** * Returns the list of cities in the database * which have a hotel * @param useless * @return the list of cities */</code></td>
</tr>
<tr>
<td>Parameter with no description</td>
<td><code>/** * Returns an hotel with specific id * @param id * @return the specified hotel */</code></td>
</tr>
<tr>
<td>Missing @return tag</td>
<td><code>/** * Returns an hotel with specific id * @param id of the hotel */</code></td>
</tr>
<tr>
<td>@return tag with no description</td>
<td><code>/** * Returns an hotel with specific id * @param id of the hotel * @return */</code></td>
</tr>
<tr>
<td>Useless @return tag</td>
<td><code>/** * @return something */</code></td>
</tr>
</tbody>
</table>

Figure 1: Mistakes in Javadoc

### 3.2 Outdated documentation

Javadoc can be considered as outdated if the code in the method changes after the documentation is written, that’s why we need to compare two versions of the project.

We retrieve a given older version of the project in the git repository, and then compare each method and its Javadoc with the current version to find the methods that have been changed but kept the same documentation.
4 Results

We tried our Javadoc Checker on a personal project to check if it works well, before executing it on real Github projects.

4.1 Finding errors

We choose a personal project, HotelApp[3], to test Javadoc Checker. As the project has a perfect Javadoc, we voluntarily add a few mistakes of each type to make our application find them. Here is an example of each warning we return for these mistakes.

<table>
<thead>
<tr>
<th>Number</th>
<th>Mistake</th>
<th>Warning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Missing Javadoc for a method</td>
<td>There is no description for method findHotelsByCity</td>
</tr>
<tr>
<td>2</td>
<td>Missing parameter in method description</td>
<td>Javadoc missing for parameter city in method List findHotelsByCity(String)</td>
</tr>
<tr>
<td>3</td>
<td>Unused parameter in method description</td>
<td>There is no parameter named uselessParameter in String[] findAvailableCities()</td>
</tr>
<tr>
<td>4</td>
<td>Useless @return tag</td>
<td>@return statement, but returnVoidMethod returns void</td>
</tr>
<tr>
<td>5</td>
<td>Missing @return tag</td>
<td>@return missing for findHotelsByCity</td>
</tr>
<tr>
<td>6</td>
<td>Parameter with no description</td>
<td>There is no description for parameter city for method List findHotelsByCity(String)</td>
</tr>
<tr>
<td>7</td>
<td>Method code has changed but not Javadoc</td>
<td>List findHotelsByCity(String) body has changed, but the javadoc is still the same</td>
</tr>
<tr>
<td>8</td>
<td>Parameter type has changed but not Javadoc</td>
<td>List findHotelsByCity(String) has changed without changing its documentation</td>
</tr>
<tr>
<td>9</td>
<td>Missing Javadoc for a class</td>
<td>Javadoc missing for class com.javaworld.hotels.model.HotelModel</td>
</tr>
</tbody>
</table>

Figure 2: Warnings for Mistakes in Javadoc

We can see that each mistake we added in the project is found, and we generate warnings which are explicit and name the location of the faulty Javadoc, so that it’s easy to correct it.
4.2 Tests on Github projects

We select five Github projects of different size to test our application. For each of them, we give its name, its Github repository and some characteristics of the current version:

<table>
<thead>
<tr>
<th>Project Name</th>
<th>number of classes</th>
<th>number of methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jfilter[5]</td>
<td>36</td>
<td>227</td>
</tr>
<tr>
<td>JSON-java[6]</td>
<td>30</td>
<td>254</td>
</tr>
<tr>
<td>ScribeJava[8]</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>SquareSquash[9]</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 3: Github Projects

We analyze each of them with Javadoc Checker, and calculate a Javadoc mistake ratio for classes and methods. If a method is concerned by several mistakes - for example, missing Javadoc implies missing parameter in method documentation - it’s only counted once.

Figure 4: Percentage of Javadoc Errors in the Github Projects in Classes and Methods Documentation
As we can see, 4 out of 5 projects have mistakes in at least 80% of the methods’ Javadoc. It means that a lot of projects have incomplete Javadoc, which is yet essential to understand a project. These results may be due in part to the fact that we consider as a mistake the lack of Javadoc in overriding methods, even if it’s normal that they just have the @Override annotation. Similarly, methods annotated with @inheritdoc, which manages documentation inheritance, are considered as erroneous but shouldn’t be. Moreover, Javadoc for getters and setters are often considered useless, but we didn’t distinguish them in all methods.

Nevertheless, the best documented project, JSON-java, has 4 times less mistakes in methods’ Javadoc than the other projects, so we can assume these four projects actually have incomplete Javadoc.

Concerning the Javadoc in classes, they are generally better documented than methods. But that may be due to the fact we just checked the existence of Javadoc, whereas for methods there are several types of error even for existing documentation. For example, we didn’t verify the presence of @author annotation in class Javadoc. But we can see that even if our verifications are less exhaustive than in methods, Jfilter still has 80% of undocumented classes, so we can assume this project really lacks in Javadoc documentation.

All things considered, JSON-java seems to be the most well documented project whereas Jfilter is the least one. As Github projects are open-source and tend to be accessible to anyone, a good documentation seems essential for a developer entering a project.

### 4.3 Analysis on error types

We define 9 typical errors in Javadoc and check if they are all found in these chosen projects. The figure 5 shows how much each type of documentation mistake represents in all the projects’ errors.

We can immediately notice that the most frequent mistakes are the lack of documentation in classes and methods. This result can be interpreted in two ways : projects are poorly documented, or when there is a documentation it’s quite complete and corresponds to the method it describes. This suggests that existing documentation tends to be well written. This also implies that the most frequent error can be found by just analyzing the current version of the project, so comparison with older versions seem to be less useful that we thought it could be.

Moreover, we can see that three types of mistake never appear in any of the five analyzed projects. First, there is no useless @return tag, which could appear if the returned type of a method was changed to void. So either this case never happens or Javadoc has always been updated. We can still notice that the opposite is not true.
since there are missing @return tags in three projects. Second, we never find Javadoc where a parameter is named but not described, which means that parameters tend to be explained when they are not missing. We can even find useless parameters in one project. Third, when comparing two versions of the project, we never find methods in which the parameter type has changed with no explanation in Javadoc. This seems quite logic as it is a big change in the method’s behavior, and as changing the type often leads to changing the name, it’s logic to update the Javadoc.

If we take the particular case of JSON-java project, which is considered as the best documented of the five, we can find a greater proportion of outdated documentation. This shows that it is quite hard to maintain a coherent documentation all along the development phase. For this kind of projects, Javadoc Checker can help determine which method’s documentation must be updated.
4.4 Limitations and Perspectives

If a developer uses our Javadoc Checker before each commit, he may be able to find all portions in his code which are not commented enough, or which contain errors after some changes. However, we admit there are some limitations in our application.

First, we compare only two versions of the project. We can easily imagine for example that a method body doesn’t change between the two versions we test, but has changed in a previous commit with no change in the corresponding Javadoc. In this case, the documentation may be outdated but Javadoc Checker won’t warn the developer as it doesn’t detect a change in code. Comparing only two versions doesn’t retrieve older problems. By comparing each method with the previous one, we could be able to find all outdated Javadoc, but currently we can only compare with the current version of the project.

Moreover, some mistakes we detect can’t really be considered as errors. For example, Javadoc documentation for getters and setters is a useless warning as it is often voluntarily omitted by developers, but we don’t distinguish them from other methods. Another case concerns overriding methods. Sometimes, just keeping the @Override annotation is enough, as in toString() overriding method for example. But it may become useful if the overriding method adds new behavior in the method, so we shouldn’t ignore it in this case. Anyway, Javadoc Checker only generates warnings, so inappropriate ones can still be ignored.

There is also another case we don’t consider in our application: if a method name changes, then it is considered as two different methods in the two versions, and outdated Javadoc is not detected. This is because methods are defined by their signature, so if the name changes, it becomes another method to Spoon, and we can’t compare the body content with the previous version.

Finally, Javadoc Checker considers a method documentation is outdated if the method code changed and not the Javadoc. However, changing source code does not always imply that its behavior has changed, especially in case of bug fixing. So it may generate a useless warning, but again it can just be ignored. An improvement could be to retrieve the commit message and check if a fixing mention exists, in this case this method could be ignored, but it here implies that commit messages contain the name of the fixed method.
5 Conclusion

We provide an automatic Javadoc Checker which is able to detect mistakes in code documentation and outputs explicit warnings. We test it on several Github projects and analyze our results. Unlike what we thought at the beginning, the most frequent mistakes are the simplest ones, such as missing Javadoc, so it’s not really useful to check for outdated documentation by comparing versions of the project, except for greatly documented ones. Some of the warnings we generate are also generated in most IDEs.

Nonetheless, Javadoc Checker can be useful for a developer who wants to participate in a Github open-source project. Indeed, it’s easier to understand and contribute to a project if it contains a good documentation. Javadoc Checker can also be useful to developers who want to improve their project’s documentation.
References