# CODES: mining sourCe cOde Descriptions from developErs diScussions

Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, Gerardo Canfora
University of Sannio, Via Traiano, 82100 Benevento, Italy
vassallocarmine@gmail.com, {spanichella, dipenta, canfora}@unisannio.it

## ABSTRACT

Program comprehension is a crucial activity, preliminary to any software maintenance task. Such an activity can be difficult when the source code is not adequately documented, or the documentation is outdated. Differently from the many existing software re-documentation approaches, based on different kinds of code analysis, this paper describes CODES (mining sourCe cOde Descriptions from developErs diScussions), a tool which applies a "social" approach to software re-documentation. Specifically, CODES extracts candidate method documentation from StackOverflow discussions, and creates Javadoc descriptions from it. We evaluated CODES to mine Lucene and Hibernate method descriptions. The results indicate that CODES is able to extract descriptions for 20% and 28% of the Lucene and Hibernate methods with a precision of 84% and 91% respectively.
Demo URL: `http://youtu.be/Rnc5ni1AAzc`
Demo Web Page:
`www.ing.unisannio.it/spanichella/pages/tools/CODES`

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

## General Terms

Documentation

## Keywords

Code re-documentation, Mining developer discussions, Program comprehension

## 1. INTRODUCTION

When evolving an existing system, or when integrating some existing code into a system under development, programmers need to properly understand source code. Depending on the specific task, such an understanding task is performed at different levels of granularity, e.g., the behavior of a method, or of a specific fragment of source code. To support code understanding, the availability of adequate documentation is important. Particularly crucial is the availability of correct, exhaustive comments, because during understanding tasks developers spend most of their time looking at source code [5]. Unfortunately, due to several reasons—such as time pressure, or the need for urgent maintenance tasks—the available documentation is scarce or outdated. For this reason, previous research has focused on several approaches aimed at re-documenting source code, for example by generating method summaries in form of Javadoc comments [6], or, at a lower level of granularity, comments describing actions performed by source code fragments [7].

While existing re-documentation approaches are mainly based on source code analysis, it might be possible to mine source code descriptions from other sources of information. Noticeably, when developers discuss over mailing lists or issue trackers, they often produce descriptions of source code elements, for example when explaining the reason why a problem occurred, or when introducing some source code to junior programmers in the context of mentoring activities [3]. For example, the following method description that can be found in the Lucene issue tracker[1]:

> *"new method added to AttributeSource: addAttributeImpl(AttributeImpl). Using reflection it walks up in the class hierarchy of the passed in object and finds all interfaces that the class or superclasses implement and that extend the Attribute interface. It then adds the interface- instance mappings to the attribute map for each of the found interfaces."*

Moreover, when a software—say a library—has been released and someone else integrates it in her code, discussion on how such a software should be used may occur on different kinds of Question and Answer (Q&A) site, e.g, StackOverflow (SO).

Based on such considerations, in a previous paper [4] we have proposed a "social" approach to software re-documentation. Specifically, we mine discussions posted on issue trackers with the aim of identifying method descriptions. Such a mining is based on heuristics aimed at matching different kinds of patterns developers use to describe methods, e.g., by explaining their syntax, their behavior in terms of other method calls, or how a method overloads/overrides other methods. A more recent, related approach is the one by Wong *et al.* [8] who developed the

[1]https://issues.apache.org/jira/browse/LUCENE-1693



*ICPC'14*, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597799

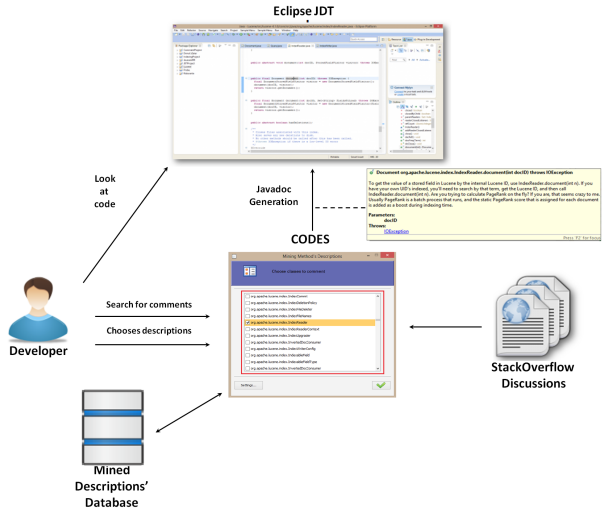



**Figure 1: CODES information flow.**

*AutoComment* tool aimed at mapping code fragments with their descriptions in SO, relying on clone detection between source code and code fragment contained in the discussion, and then automatically generating comments. Our approach differs from that of Wong *et al.* [8], because we aim at mining method descriptions instead of code fragment descriptions; for this reason, we also do not assume that the discussion contained a pertinent code fragment.

In this paper we describe CODES (mining sourCe cOde Descriptions from developErs diScussions), an Eclipse plugin that automatically extracts Java method descriptions from SO discussions. The plugin implements a description-mining approach defined in our previous work [4], which was originally conceived to mine method descriptions from mailing lists and issue tracker discussions. The approach has been adapted to mine SO discussions too, plus it is now able to group together multiple candidate discussions for the same artifact (and remove duplicates). CODES can be used for re-documentation purposes in two different scenarios: (i) the original developers can rely online discussion to better re-document their own code, or (ii) a system integrator downloads some scarcely documented third-party source code, and needs to understand it.

**Paper structure.** Section 2 briefly describes the method description mining approach proposed in our previous paper [4] and its implementation as an Eclipse plugin. Section 3 describes how CODES works, while Section 4 provides some info about the accuracy and completeness of the mining performed by CODES. Finally, Section 5 concludes the paper.

## 2. OVERVIEW OF THE APPROACH AND ITS IMPLEMENTATION IN ECLIPSE

This section summarizes the approach behind CODES. Further details can be found in a previous paper [4].

Figure 1 depicts the CODES Eclipse plugin flow of information. First, the developer selects the list of classes that she wants to re-document. Then, Java Reflection API is used to perform an introspective analysis of the classes to be re-documented, with the aim of retrieving information about its methods, i.e., parameters, return value and the name of the method. After the introspective analysis, CODES uses the SO search engine to search the set of discussions that describe a given Java method, using as search keys project name + class name + method name. More precisely, CODES passes the search key to SO through its REST interface, i.e., through the URL[2]. Then, the SO search engine returns back the URLs of the of discussions relevant to the formulated query. Such discussions are composed of *Questions* and *Answers*.

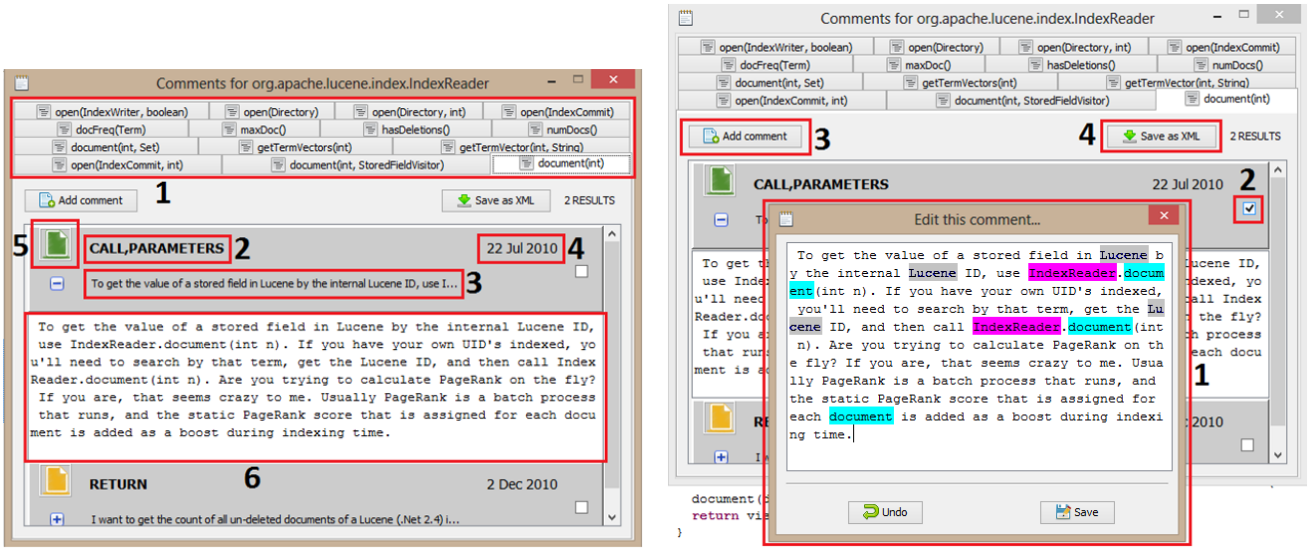
By using regular expressions, CODES checks whether the *Questions* contain the name of the project and discards the discussions related to other projects. Note that for this purpose we do not rely on SO tags exclusively, because they are not used consistently. After that, the set of URLs is further restricted to discussions traced to the class to be re-documented. This is done using an approach inspired by the one proposed by Bacchelli *et al.* [1], i.e., it works by matching the class name (or fully qualified class name when possible) in the discussion.

Then, CODES processes paragraphs contained in the SO *Answers*. In particular, we consider answers that are voted by at least one SO user. To this end, we search for paragraphs that contain words matching names of methods of the classes to be documented. We are aware that this does not guarantee yet that the paragraph describes such a method, also because the name could have been matched by chance. After having mapped paragraphs onto methods, we classify them onto three categories of candidate method descriptions:

- *Syntactic descriptions*: this heuristic aims at identifying paragraphs that describe method syntactic descriptions. Such paragraphs contain (i) at least a given percentage $s_1$ of the method parameter names if the method has input parameters (i.e., paragraphs explain the method inputs, or at least some of them), and (ii) the word "return" if the method is not *void*. As explained in the previous paper, we have set $s_1$ to 50%[4] because analyzing the distribution of parameters referred to in the paragraphs traced onto methods the median is equal to 50% for all the projects considered in that study[4].
- *Method invocations*: this heuristic aims at identifying paragraphs that describe a method behavior in terms of invocations of other methods. Such paragraphs must contains (i) words among "call", "execute", and "invoke", and (ii) must mention at least one of the methods invoked by the method being documented.
- *Overriding/Overloading*: this heuristic aims at identifying paragraphs that describe how a method overrides/overloads another. After checking if, indeed, the method being re-documented does an overload or override, we check the presence of the words "overload" or "override" in the paragraph.

Paragraphs that do not fall in any of the above three categories are discarded. Since CODES can potentially associate a discussion to more than one class, to increase the accuracy of the approach, we apply a further step, in which we compute the textual similarity between the candidate paragraphs identified in the previous step and the body of the methods they re-document. The similarity is computed

[2]http://stackoverflow.com/search?q=search keys




(a) Mined method Descriptions.

(b) Description editing (relevant keywords highlighted in different colors).

**Figure 2: Browsing the mined descriptions.**

by indexing both source code and textual paragraphs using a Vector Space Model [2]. After having performed stop word removal, Snowball stemming, and *tf-idf* term weighting, we compute the cosine similarity between candidate descriptions and method corpora, and discard descriptions for which the cosine is smaller than a threshold $th_T$ (calibrated to 0.4 in our previous work [4]). After such a filtering, we use again textual similarity to prune out duplicate descriptions of the same method, i.e., paragraphs describing the same method and having a cosine similarity greater than 0.8. Finally, the obtained descriptions are stored in a local database implemented using SQLite. Then, the user can browse such descriptions from a window in the IDE, modify them, and if needed automatically inserting them in the code in form of Javadoc comments.

# 3. CODES IN ACTION

This section describes how CODES works, explaining its main features.

## 3.1 Starting to Search for Candidate Descriptions

The developer starts to use CODES by right-clicking on the Java project and selecting, in the project's context menu, the option *"Mining Method Descriptions"*. CODES opens a Window that allows the developer to select the class(es) that she wants to re-document. When the developer clicks on the "Confirm" button, CODES starts searching for method descriptions of the selected class(es). Before starting the search for candidate descriptions, the developer can click the button *"Settings"* to access in the search configuration window. This allows the developer to choose between two possible sources of descriptions: (i) a local database containing descriptions downloaded from SO during previous online searches, (ii) direct online queries to SO.

Once the developer clicks on the *"Confirm"* button, CODES starts to search for descriptions in SO and shows, with a progress bar, the status of the search process. Once the search has been completed, the developer can start browsing/analyzing the candidate descriptions found.

## 3.2 Browsing and Editing Candidate Descriptions

CODES generates a frame like the one shown in Figure 2(a) for each class for which candidate method descriptions were found in SO or in the local database. The frame contains, for each method, a tab (point 1, Figure 2(a)) with expandable panels, reporting the descriptions found for that method. In addition, each panel reports: (i) the description types (point 2, Figure 2(a)) i.e., whether it is a syntactic description, behavior description, or overload/override description, (ii) a description preview (point 3, Figure 2(a)), the date when the description was posted on SO (point 4, Figure 2(a)), and an icon (point 5, Figure 2(a)) colored green or yellow depending on the relevance of the description itself. CODES allows a developer to expand the panel and browse all descriptions for a given method (point 6, Figure 2(a)). By clicking on any of these descriptions, CODES opens a new frame (point 1, Figure 2(b)), which provides the possibility to edit the description text. In such a view, relevant keywords are highlighted using different colors: (i) matched method name in light blue, (ii) matched project name in gray, and (iii) matched class name in purple. From this view, the developers can also export the description(s) in XML by clicking on the *"Save as XML"* button (point 4, Figure 2(b)). For example, we used such option for the manual validation of the paragraphs extracted, as described in Section 4.

## 3.3 Adding Mined Descriptions as Javadoc Comments

CODES allows the developer to select (point 2, Figure 2(b)) one or more descriptions to build a Javadoc comment for a specific method. Once the desired descriptions, the


**Figure 3: Generating a Javadoc comment from mined descriptions.**

developer can click on the *"Add Comment"* button (point 3, Figure 2(b)) to generate the Javadoc comment, which will be automatically inserted in the code browseable from the Eclipse-JDT IDE (point 1, Figure 3).

## 4. PERFORMANCE EVALUATION

In our previous work [4] we evaluated the approach implemented in CODES by mining method descriptions of Lucene and Eclipse in the issue trackers of these two projects. The proposed approach selected as good candidate method descriptions 3,111 paragraphs for Eclipse and 3,707 for Lucene, covering 36% of the methods for Lucene and 7% for Eclipse. As pointed out in our previous paper, although such percentages appear to be low, they are reasonable, because it is unlikely to find, in developers' communication, a thorough description of all possible methods. A manual validation on a sample of 250 candidate descriptions for each projects indicated a precision of 79% for Eclipse and of 87% for Lucene.

To evaluate the CODES plugin when mining description from SO, we considered the developers discussions in SO related to Lucene 2.9.0 (September 2009) and Hibernate 3.5.0 (August 2009), and applied a similar empirical evaluation that we performed in the previous work. In particular, CODES found candidate method descriptions for 20% of the Lucene's methods and 28% for the Hibernate's methods. A manual validation of 100 of the 9,343 descriptions mined for Lucene and 100 of the 10,608 descriptions mined for Hibernate indicated a precision of 84% for Lucene and 91% for Hibernate.

## 5. CONCLUSIONS AND WORK-IN-PROGRESS

This paper presented CODES, an Eclipse plugin to automatically extract Java method descriptions from discussions in SO. The tool is based on a "social" approach defined in our previous work [4] and adapted to mine SO. CODES searches SO for method descriptions of selected Java classes, and then recommends to the developers, who can edit such descriptions, and then ask CODES to generate Javadoc comments. The tool can be used by developers/owners of an open source project to re-document their own code or, if needed, by integrators to re-document and understand poorly commented open source code they want to reuse and integrate in their projects. CODES is currently available for download[3], together with a video explaining its features through a demonstration scenario. Future work aims at enhancing CODES improving its features in terms of usability and adding new features, e.g., for re-documenting classes or packages.

## References


[1] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 375–384. ACM, 2010.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[3] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. Who is going to mentor newcomers in open source projects? In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-12), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 44. ACM, 2012.

[4] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Proceedings of the 20th IEEE International Conference on Program Comprehension*, pages 63–72, Passau, Bavaria, Germany, 2012. IEEE.

[5] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265. IEEE Press, 2012.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 43–52. ACM, 2010.

[7] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 101–110. ACM, 2011.

[8] E. Wong, J. Yang, and L. Tan. AutoComment: Mining question and answer sites for automatic comment generation. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Palo Alto, USA, November 11-15, 2013*, 2007.


[3] `www.ing.unisannio.it/spanichella/pages/tools/CODES`