Mining Source Code Descriptions from Developer Communications

Sebastiano Panichella†, Jairo Aponte‡, Massimiliano Di Penta†, Andrian Marcus⋆, Gerardo Canfora†
† Dept. of Engineering-RCOST, University of Sannio, Italy
‡ Universidad Nacional de Colombia, Bogota, Colombia
⋆ Wayne State University, Detroit USA

***Abstract*—Very often, source code lacks comments that adequately describe its behavior. In such situations developers need to infer knowledge from the source code itself or to search for source code descriptions in external artifacts.**

We argue that messages exchanged among contributors/developers, in the form of bug reports and emails, are a useful source of information to help understanding source code. However, such communications are unstructured and usually not explicitly meant to describe specific parts of the source code. Developers searching for code descriptions within communications face the challenge of filtering large amount of data to extract what pieces of information are important to them. We propose an approach to automatically extract method descriptions from communications in bug tracking systems and mailing lists.

We have evaluated the approach on bug reports and mailing lists from two open source systems (Lucene and Eclipse). The results indicate that mailing lists and bug reports contain relevant descriptions of about 36% of the methods from Lucene and 7% from Eclipse, and that the proposed approach is able to extract such descriptions with a precision of up to 79% for Eclipse and 87% for Lucene. The extracted method descriptions can help developers in understanding the code and could also be used as a starting point for source code re-documentation.

***Index Terms*—Code re-documentation, mining e-mails, program comprehension.**

I. Introduction

Consider the following situation. A developer is reading the Java code of an unfamiliar (part of the) system. She encounters a methods call. Ideally, a good method name would indicate its purpose. If not, a nice Javadoc comment would explain what the goal of the method is. Unfortunately, the method name is poorly chosen and there are no comments. Not an uncommon situation. At this point, the developer has the choice of reading the implementation of the method or searching the external documentation. It is very rare that external documentation is written at method level granularity (especially when comments are missing) and that such specific information is easy to retrieve. The goal of our work is to help developers in such situations. Specifically, we aim at providing developers with a means to quickly access descriptions of methods.

Our conjecture is that, if other developers had any issues related to a specific method, then a discussion occurred and someone described the method in the context of those issues. For example, developers and project contributors communicate with each other, through mailing lists and bug tracking systems. They often "instruct" each other about the behavior of a method. This can happen in at least two scenarios. First, when a person (sometimes a newcomer in the project) is trying to solve a problem or implement a new feature, she does not have enough knowledge about the system, and asks for help. Second, when a person explains to others the possible cause of a failure, illustrating the intended (and possibly also the unexpected) behavior of a method. For example, we report a paragraph for issue #1693 posted on the Lucene Jira bug-tracking system[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."

which clearly describes the behavior of the *AttributeSource: addAttributeImpl(AttributeImpl)* method.

We claim that *unstructured communication between developers can be a precious source of information to help understanding source code.*

So, why developers could not use simple text search techniques, based on text/regular expression matching utilities, such as, *grep*, to find method descriptions in communication data? Such simple text matching approaches could only identify sentences having a method name, or in general any regular expression containing the method name plus other strings such as the class name or some parameter names. They would generate too many false positives. As it happens for requirement-to-code traceability recovery [1], [2], such a simple matching is not enough.

This paper presents and validates an approach to automatically mine source code descriptions—in particular method descriptions—from developer communications, such as, emails and bug reports[2]. It also presents evidence to support our assumption that developer communications are rich in useful code descriptions.

Our approach traces emails to classes, identifies affirmative textual paragraphs in these emails, and traces such paragraphs to specific methods of the classes. Then, it uses heuristics—

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

[2] For simplicity, we will only refer to mailing lists/emails, although the approach is applicable to bug tracking systems and other similar communications. Only where it matters we will refer to mailing lists and bug tracking systems separately.

978-1-4673-1216-5/12/$31.00 © 2012 IEEE

based on textual similarity between paragraphs and methods, and on matching method parameters and other keywords to paragraphs—to filter out candidate method descriptions.

The filtering technique results in a set of one or more paragraphs describing each method (for which a description was found). These paragraphs may overlap, in terms of content, or they could describe different aspects of the method behavior, e.g., one describes the method interface and return value, another the behavior in terms of calls to other methods, another the exceptional behavior, etc. The technique retrieves all these paragraphs and combines them into a method description. Such descriptions can have multiple uses:

1) They can be used as such to help developers understanding the code.
2) In perspective, an automatic tool can further process the descriptions and automatically generate method documentation, e.g., API descriptions or comments.

We have applied the proposed approach to 26,796 bug reports from the Eclipse project, and 18,046 emails and 3,690 bug reports from the Apache Lucene project. Results indicate that emails and bug reports contain descriptions for about 7% of the Eclipse methods and 36% of the Apache Lucene methods. The proposed filtering approach is able to correctly identify method descriptions in 79% of the cases for Eclipse and 87% of the cases for Lucene. Finally, we report several examples describing how methods are likely described in the developers' communication, discussing the linguistic patterns we found in Eclipse and Lucene for different kinds of method descriptions.

The paper is organized as follows. Section II describes the proposed approach. Section III reports the empirical evaluation using data from Eclipse and Apache Lucene, while Section IV discusses some example of method descriptions found by the approach. Section V discusses the related work, while Section VI concludes the paper and outlines directions for future work.

II. Mining Method Descriptions from Communications

This section describes the proposed approach for mining method descriptions in mailing lists or bug reports.

A. Step 1: Downloading Emails and Tracing them onto Classes

First, we download mailing list archives and all bug reports concerning the analyzed time period of the investigated projects. Then, we extract the body from emails using a Perl mailbox parser (*Mail::MboxParser*). Bug reports (downloaded in HTML) are first rendered using a textual browser *lynx* and then the text is extracted using a Perl script. Then, we trace emails onto source code classes (referring to the system release before the email date). For this purpose, we use two heuristics:

1) We use an approach similar to the one proposed by Bacchelli *et al.* [3], [4]. More specifically, we assume there is an explicit traceability link between a class and an email whenever (i) the email contains a fully-qualified class name (e.g., *org.apache.lucene.analysis.MappingCharFilter;*); or (ii) the email contains a file name (e.g., *MappingCharFilter.java*)—provided that there are no other files with the same name, or that the file name is also qualified with its path.
2) For bug reports, we complement the above heuristic by matching the bug ID of each closed bug to the commit notes, therefore tracing the bug report to the files changed in that commit (if any are found).

B. Step 2: Extracting Paragraphs

During a preliminary investigation we determined—by inspecting emails from out case studies—that paragraphs describing different aspects of the email topic are separated by one or more white lines. Therefore, we use such heuristics to split each email into paragraphs. For bug reports, different posts related to the same bug report are treated as separated paragraphs.

Emails often contain source code fragments and/or stack traces that should be pruned as we are interested to mine descriptive text only (in future, we plan to keep such code fragments into account to better link paragraphs to source code). To remove them, we used an approach inspired to the work of Bacchelli *et al.* [5]. We computed, for each paragraph, the number and percentage of programming language keywords and operators/special characters (e.g., curly braces, dots, arithmetic and Boolean operators, etc.). Paragraphs containing a percentage of keywords and special characters/operators higher than a given threshold are discarded. Calibrating such a threshold requires a trade-off. We adopt a conservative approach and are willing to accept losing a few good paragraphs. Similarly to what also shown in the paper by Bacchelli *et al.* [5], we found a threshold of 10% to be the best compromise between losing good paragraphs and keeping source code fragments. Remember that our goal is to provide precise descriptions to the developer in order to save time and effort.

A further processing—performed using the English Stanford Parser[3] [6]—aims at preserving only paragraphs in the affirmative form, removing those in interrogative forms—because we assume that method description should not contain interrogative sentences—as well as pruning out sequences of words that the parser was not able to analyze, i.e., sequences of words that cannot be considered valid English sentences.

C. Step 3: Tracing Paragraphs onto Methods

To trace paragraphs onto methods, we first extract signatures for all methods in a system version released before the email being analyzed. This is done using the Java *reflection* API.

Then, we identify the paragraphs referring a method. These paragraphs shall meet the following two conditions:

- They must contain the keyword "method". This is because we are searching sentences like "The method foo() performs...". Indeed, there could be cases where the

[3]http://www-nlp.stanford.edu

method is referred and described in a sentence not containing the keyword "method" (e.g., "foo() performs..."). However, we observed in a preliminary analysis that such cases occur mostly when the method is mentioned in other contexts (e.g., describing a fault) rather than when communicating a method description to other people.
- They must contain a method name, among the methods of classes traced to the email in *Step 1*. We also require that such a name must be followed by a open parenthesis—i.e., we match "foo(" while we do not consider "foo". This is to avoid cases when a word matches a method name, while it is not intended to refer to the method. For example, we found several paragraphs like that e.g., "Method patch", where "patch()" was actually a method of a class traced onto the email.

It is important to note that such a process can be subject to ambiguities. First, an email can be traced onto multiple classes, having one or more method with the same name (and maybe even the same signature). In such a cases, the paragraph is assigned to all of these classes. Second, there may be overloaded methods. Where possible, this is resolved by comparing the list of parameter names mentioned in the paragraph with the list of parameters in the method signature as extracted from the source code. When this is not sufficient to resolve the ambiguity, we conservatively assign the paragraph to all matched methods. As explained later (*Step 5*) both ambiguities can be mitigated by computing the textual similarity between the paragraph and the method.

D. Step 4: Filtering the Paragraphs

We defined—based on the manual inspection of hundreds of emails— a set of heuristics to further filter the paragraphs associated with methods.

These heuristics encode some observed rules of discourse commonly used by developers in emails. The first heuristic concerns *method parameters*: it is required that, if a method has parameters, at least some of them are mentioned in the method description. We count the number and percentage of method parameter names mentioned in the paragraph. We define a score, s_1 as the ratio between the number of parameters mentioned and the total number of parameters in the method. We consider $s_1 = 1$ if the method does not have parameters.

We defined three additional heuristics that captures characteristics of three different categories of method descriptions, i.e., syntactic descriptions, description of how a method overloads/overrides another one, and descriptions of how a method performs its task by invoking other methods.

1) *Syntactic descriptions (mentioning return values)*: if a method is not *void*, we check whether the paragraph contains the keyword "return". We define a score $s_2 = 1$ if the method is *void*, or if is not *void* and the paragraph contains "return", zero otherwise.
2) *Overriding/Overloading*: keywords such as "overload" or "override" are likely to be contained in some paragraphs describing methods. This, in particular, happens when a paragraph describes the additional behavior with respect to the overridden/overloaded method. We define a score $s_3 = 1$ if any of the "overload" or "override" keywords appears in the paragraph, zero otherwise.
3) *Method invocations*: when a paragraph describes a method, often it describes it in terms of invocation of other methods. Therefore, we mine the paragraphs containing for the words "call", "execute", "invoke" (or their plurals/conjugations). We define a score $s_4 = 1$ if any of the "call", "execute", or "invoke" keywords appears in the paragraph at least once, zero otherwise.

We apply the above described heuristics by constraining the set of selected paragraphs such that $s_1 \geq th_P$ and $s_2 + s_3 + s_4 \geq th_H$, where th_P is a threshold we set for the parameter heuristic and th_H is a threshold for the other heuristics. Details about the two thresholds are reported in Section III-A.

E. Step 5: Computing Textual Similarities Between Paragraphs and Methods

After having filtered paragraphs using the heuristics, we rank them based on their *textual* similarity with the method they are likely describing. The rationale is that, other than the method name, parameter names, and other keywords identified in *Step 4*, such paragraphs would likely contain other words (e.g., names of invoked methods, variable names, local variables, etc.) contained in the method body. Also, as mentioned above, computing such a similarity would help mitigating ambiguities when tracing paragraphs onto methods.

To this aim, we extract the method's text from the system source code (again, referring to a version before the email). This is done using the *srcml* analyzer [7]. Then, we normalize the method text removing special characters, English stop words, and programming language keywords, and splitting the remaining words using the camel case heuristics. A similar text pruning is performed on paragraphs. After that, we index the paragraphs and the methods using a Vector Space Model implemented with the R[4] *lsa* package.

We compute the textual similarity between each paragraph P_k and the text of each traced method M_i using the cosine similarity [8]. For each method M_i, we rank its relevant paragraphs P_k by the similarity measure. Finally, we consider only the paragraphs that have a similarity measure higher than a threshold th_T. These are the paragraphs that are presented to the user as containing a description to the method M_i. As it will be shown in Section III-C, varying th_T will produce different results in terms of precisions and of retrieved candidate method descriptions.

F. Limitations of the Proposed Approach

Our proposed approach—to the best of our knowledge—represents the first attempt to mine method descriptions from developers' communication. As with any work that addresses a problem in premiere, limitations exist, which we hope to address in future work. We highlight here those that should

[4]http://www.r-project.org

be kept in mind while interpreting the results of our empirical evaluation from the next section:

- *We do not consider sequences of paragraphs that can describe the same method.* In some cases, a method description can be longer than one paragraph and thus span over multiple paragraphs. A preliminary attempt at clustering subsequent paragraphs to the ones that mention a method drastically lowered the precision of our approach. More sophisticated approaches are needed to address this issue.
- *Paragraphs often describe partial/exceptional behavior.* In some cases, the paragraphs describe only part of the method behavior, because the communication concerns only that part. In other cases, the paragraphs describe exceptional behavior. We believe that such paragraphs are useful to the developers to get a complete or partial overview of a method's syntax and behavior.
- *We do not really mine abstractive method description, but rather extractive descriptions.* Since our technique uses textual similarities, it will recover paragraphs describing a method behavior only if this is done using (some of) the elements (e.g., names of invoked methods, method parameters, local variables, etc.) contained in the method body—a.k.a, extractive description. For example, our approach may not recover a paragraph providing a high-level description of an algorithm (e.g., imagine a paragraph describing a sorting algorithm), which uses terminology not used in the implementation of the method—a.k.a., abstractive description. However, mixed descriptions will likely be retrieved by the approach.

III. Empirical Evaluation

The *goal* of this study is to evaluate the proposed approach for extracting method descriptions from developers' communications. The *quality focus* is the ability of the proposed approach to cover methods from the analyzed systems, as well as the precision of the proposed approach. The *perspective* is of researchers who want to evaluate to what extent mining developers' communications can be used to support code understanding and to what extent the proposed approach is able to identify method summaries with a reasonable precision. The *context* consists of bug reports from the Eclipse project and both mailing lists and bug reports from the Lucene project. Eclipse[5] is an open-source integrated development environment, written in Java. Lucene[6] is a text retrieval library developed in Java. Table I reports some relevant characteristics of the two systems and the data we used. While Eclipse can be considered as a large system, Lucene is a small-medium system.

The empirical study reported in this section aims at addressing the following research questions:

- **RQ1** *How many methods from the analyzed software systems are described by the paragraphs identified by the proposed approach?* While we do no expect to find descriptions for all, or nearly all of the methods, we believe that the approach would be useful in the practice only if finding descriptions for a given method would not be an extremely rare event.
- **RQ2** *How precise is the proposed approach in identifying method descriptions?* This research question aims at determine whether the mined description are meaningful method descriptions, or whether they are, instead, *false positives*. While some false positives are unavoidable, too many of them would make the approach unpractical.
- **RQ3** *How many potentially good method descriptions are missed by the approach?* This research question aims at providing an idea of how the proposed approach is affected by *false negatives*, i.e., filtering out good method descriptions.

TABLE I
Characteristics of the two subject systems.

Characteristic	Eclipse	Lucene
Analyzed Period	2001–2010	2001–2011
KLOC range	283-2,486	6–345
#of classes (range)	4,829–18,834	427–528
#of methods (range)	31,132–117,654	2,432–2,952
# of bug reports	26,796	3,690
# of emails	–	18,045
# of paragraphs from bug reports	202,539	115,504
# of paragraphs from emails	–	91,408
Total # of paragraphs	202,539	206,912

A. Threshold Calibration

Step 4 of the proposed approach relies on two thresholds, th_P and th_H. To calibrate th_P, we analyze the distribution parameters referred to in the paragraphs traced onto methods. For Eclipse, the percentage of parameters had a minimum and first quartile equal to zero, a median=50%, and a third quartile and maximum equal to 100%. We selected the median as threshold and analyzed the performance with different settings for th_P varying it between 0% and 100% in 10% steps. We confirmed that the median choice works equally well for Lucene. We realize that such a rule for selecting th_P cannot be easily generalized, but it worked for these two systems and for proof of concept purpose. Investigating the generality of this rule is subject of future work.

Regarding th_H, we set it to 1—i.e., accepting all cases where $s_2 + s_3 + s_4 \geq 1$—in order to select paragraphs containing at least one keyword able to characterize the paragraph with respect to the different kinds of method descriptions outlined in *Step 4* of the approach. Once again, identifying alternative rules for calibration, which generalize better, is subject of future work.

The effect of the third threshold, th_T, on the precision of the approach is analyzed in detail in the following subsection.

B. Evaluation Procedure

First, we extracted, using *Steps 1-3* of the proposed approach, a set of candidate paragraphs that are traced onto methods. We refer to them as the subset of *traced paragraphs*.

[5]http://www.eclipse.org
[6]http://lucene.apache.org

After that, we performed a first pruning using the heuristics from *Step 4*, i.e., selecting all paragraphs—referred to as *candidate descriptions* from here on—having $th_P \geq 0.5$ and $th_H \geq 1$.

Subsequently, we computed the cosine similarities, with the aim of investigating how the performance of the approach varies by considering only paragraphs having a cosine similarity greater than a given threshold.

Then, we built the oracle against which to validate our results. The oracle was done by manually validating a sample of the *candidate descriptions*. Since it was not possible to manually validate all descriptions, we sampled 250 descriptions for each project. Such a sample allows to achieve estimations with a confidence interval of $\pm 5\%$ assuming a significance level of 95% [9]. We decided not to perform a random sampling of the descriptions: since our aim is to analyze how the precision and the method coverage vary with different thresholds of the cosine similarity, we wanted to include in the sample enough data points representative of different cosine ranges. Therefore, the most appropriate way to proceed was to apply a stratified sampling. We divided our population of candidate descriptions in sets according to five classes of cosine range: 0%-20%, 20%-40%, 40%-60%, 60%-80%, and 80%-100%. Then, based on the distribution of descriptions over the different classes, we randomly sampled 25, 50, 100, 50, 25 descriptions for the five classes, respectively.

Then we asked three reviewers to analyze the sampled descriptions and decide whether they were, or not, reasonable paragraph descriptions. Two reviewers were two of the paper authors (one of which did not know the detail of the mining algorithm at the time the validation was performed, so not to bias such a validation), and the third reviewer was a PhD student not involved in the work. To rate a description, the three reviewers had the system source code available and checked whether the description is, indeed, one possible way a method could be described, either in terms of its syntax, as extension of other methods, or in terms of a method invocation chain. If all three reviewers agreed that a paragraph is a specific kind of description for a method, then the paragraph was classified as true positive. If all three reviewers agreed that a paragraph is not a good description for a method, then the paragraph was classified as false positive. When they disagreed, they discussed until they reached consensus. In the end, **500** paragraphs were included in the oracle.

To address **RQ1** we considered all the methods in the analyzed systems, whereas for **RQ2** we only used the paragraphs in the oracle, in order to analyze how the *method coverage* (**RQ1**) and the *precision* (**RQ2**) change when increasing the cosine threshold. We define the *method coverage* for a given cosine threshold th_T as the percentage of the methods in the system for which there exists at least one candidate description traced onto it and such that $cos(m_i, P_j) > th_T$. We define the *precision* for a given cosine threshold th_T as the percentage of true positives in the oracle for which $cos(m_i, P_j) > th_T$.

Addressing **RQ3** is more difficult. We are aware that precisely assessing false negatives would be impossible (it would require analyzing the entire body of emails). Instead, we extracted a small sample (100 paragraphs for each project, thus in total further 200 paragraphs) from the set of paragraphs pruned after applying the *Step 3* heuristics, i.e., all paragraphs that can be mapped onto a method (and not all possible paragraphs, because we assume that a paragraph describing a method at least mentions it), but do not satisfy our similarity-based filtering. We manually validated the sample similarly to how we did it for the oracle, in order to compute the percentage of *false negatives* in the sample.

TABLE II
NUMBER OF PARAGRAPHS AND METHOD COVERAGE AFTER APPLYING FILTERING FROM STEPS 2, 3 AND 4 OF THE APPROACH.

Filtering	Eclipse		Lucene	
	# of paragraphs	method coverage	# of paragraphs	method coverage
Step 2	202,539	–	206,912	–
Step 3	42,095	22%	12,417	65%
Step 4	3,111	7%	3,707	36%

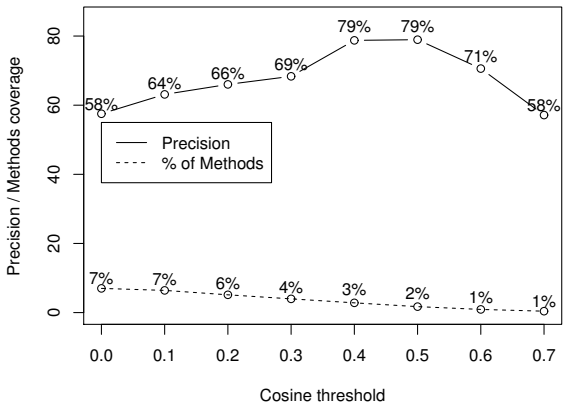
C. Results

Table II reports results about the number of paragraphs obtained after applying steps 2, 3, and 4 of the approach, i.e. (2) the initial set of paragraphs extracted from the emails after pruning out source code and irrelevant sentences (short and interrogative ones), (3) the number of paragraphs traced to methods, and (4) the number of paragraphs traced to methods that satisfy the filtering according to *Step 4* heuristics i.e., paragraphs with $th_P \geq 0.5$ and $th_H \geq 1$. The table also reports, for the last two cases, the percentage of covered methods. As it can be noticed, about 20% of the Eclipse paragraphs and 5% of the Lucene paragraphs can be traced to methods (*Step 3*), which ensures a coverage of 22% of the Eclipse methods and 65% of the Lucene methods. However, such paragraphs do not satisfy the heuristics of *Step 4*, nor they are constrained by any textual similarity threshold.

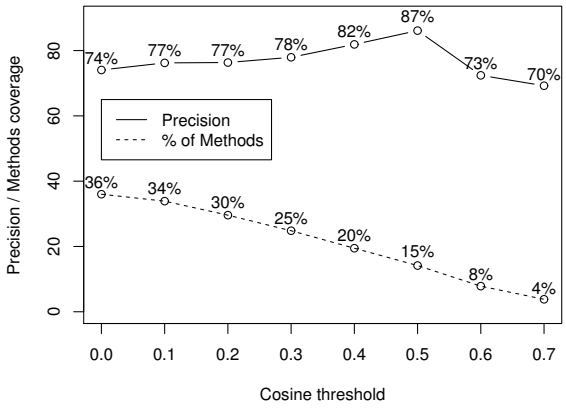
When applying the heuristics of *Step 4*, the number of paragraphs is reduced to 3,111 for Eclipse and 3,707 for Lucene, which results in 7% method coverage for Eclipse and 36% for Lucene.

Fig. 1 reports the achieved precision and the method coverage for both Eclipse and Lucene. The x-axis shows the increasing cosine similarity threshold, th_T, while the y-axis shows both the precision and the method coverage. Note that the precision is on a 0-100% scale, whereas maximum coverage for Eclipse is 22% and for Lucene is 65%.

The results for both systems correspond to our expectations, i.e., increasing the cosine threshold results in an increase in precision and it comes at the cost of reduced method coverage. An interesting phenomenon is that the increase in precision peaks (79% for Eclipse and 87% for Lucene) at a threshold of approximately 0.5 for both systems, which means that maximum precision can be achieved without complete loss of method coverage. In both cases, the difference between the minimum and maximum precision is higher than the difference between the minimum and maximum method coverage



(a) Eclipse



(b) Lucene

Fig. 1. Precision and method coverage for different levels of similarity. Note that the maximum possible coverage would be (see Step 3 of Table II) 22% for Eclipse and 65% for Lucene.

(proportionally). In other words, the precision gain increases slower than the loss in method coverage. Method coverage in Eclipse for highest precision is about 3% (which means covering between 933 and 3,530 methods, depending on the version), and for Lucene is about 15% (which means covering between 365 and 442 methods).

We can summarize the results related to **RQ1** (method coverage) and **RQ2** (precision) stating that, on the one hand, the proposed approach is precise enough to mine method descriptions, thus reducing the developers' burden to go through a wide number of false positives. On the other hand, the percentage of covered methods could appear as relatively low, thus it is useful to pursue a compromise between coverage and precision. However, it is important to note that (i) we cannot really expect to find descriptions for all methods, especially for large systems like Eclipse (for which, by the way, we do not have emails, but bug reports only), and (ii) the coverage depends a lot on the quality—with respect to our goal of mining descriptions—of the project discussion, which in our case seems to be better for Lucene than for Eclipse.

Regarding **RQ3**—i.e., the presence of *false negatives*—the analysis of a sample of 100 paragraphs traced to methods, but not satisfying the *Step 4* heuristic, indicates that, for Eclipse, 78 out 100 paragraphs have been classified as *true negatives*, leaving 22 paragraphs that could represent good method descriptions, but that were discarded by our heuristics. For Lucene, 67 paragraphs were classified as true negatives, leaving a relatively large (33%) number of *false negatives*. Although this can be seen as a limitation of the proposed approach for capturing good method descriptions, this can be explained by the peculiar characteristics of the Lucene mailing lists and bug reports, which contain many very good method descriptions, as it has also been noticed from the high precision obtained for **RQ2**. As stated before, heuristics that result in a better balance between a low number false negatives and a low number false positives will be investigated in the future. The current results are encouraging enough to motivate future research.

D. Threats to Validity

This section describes the main threats to validity that can affect the evaluation of our results. Given the kind of validation performed, it is worthwhile to mainly discuss threats to *construct* and *external* validity.

Threats to *construct validity* mainly concern, in this context, the measurements used in the evaluation. First, we are aware that, for assessing precision, we sampled only a subset of the extracted descriptions. However, (i) the sample size limits the estimation imprecision to $\pm 5\%$ for a confidence level of 95%, and (ii) to limit the subjectiveness and the bias in the evaluation, three evaluators (one not involved in the paper and one not knowing the details of the approach) manually analyzed the sample. Another threat to construct validity concerns **RQ3**. As explained in Section III-B, it is always difficult to perform a thorough assessment of *false negatives*. To deal with such a threat we evaluated a sample of 100 paragraphs not detected by the proposed heuristics. The actual number of false negatives in the entire system may be different than in the random sample.

Threats to *external validity* concern the generalization of results. We must remember that the main aim of this paper is to investigate whether mailing lists and bug tracking systems are a useful source of information for understanding and potentially re-documenting source code, and to propose a novel approach to mine such descriptions (at proof of concept level), rather than to perform a thorough evaluation. The empirical evaluation here is limited to mailing lists/bug reports from two systems only. Clearly, it is important to point out that variables such as the project domain, the availability of mailing lists and bug reports (as well as their quality) could influence the performance of the proposed approach. Therefore, a more extensive evaluation with data sets from further systems is

TABLE III
EXAMPLES OF TRUE POSITIVE PARAGRAPHS FOR ECLIPSE.

Class	Method	Paragraph
ServiceLoader	ServiceLoader	Similarly to osgi services, the java serviceloader takes the name of the class for which you want a service. In the present case, we want an instance of the JavaCompiler service, so the actual call being made is: ServiceLoader.load(javax.tool.JavaCompiler) This method returns an iterator on all the services available.
Wizard	addPages	In the particular case of the NewLocationWizard, you should be able to get around it by creating a protected createMainPage method which you can override in the subclass. You can then call super.addPages() from the subclass (Wizard) add pages to avoid the duplication of the setting of the properties. I still don't think that "alternative" is the proper term to use everywhere.
GC	drawString	The -1 value for bidiLevel is correct since it indicates that you're not using bi-directional text. As Randy mentioned, this might be a GDI+ issue that got introduced in 3.5. Create an SWT GC, invoke setAdvanced(true), and then use its drawString() method to draw some text in your language. Also try drawText().

TABLE IV
EXAMPLES OF TRUE POSITIVE PARAGRAPHS FOR LUCENE.

Class	Method	Paragraph
AttributeSource	addAttributeImpl	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. AttributeImpl now has a default implementation of toString that uses reflection to print out the values of the attributes in a default formatting.
Scorer	score	This proposes to expose appropriate API on Scorer such that one can create an optimized Collector based on a given Scorer's doc-id orderness and vice versa. QueryWeight implements Weight, while score(reader) calls score(reader, false /* out-of-order */) and scorer(reader, scoreDocsInOrder) is defined abstract. One other optimization is to expose a topScorer() API (on Weight) which returns a Scorer that its score(Collector) will be called, and additionally add a start() method to DISI. That will allow Scorers to initialize either on start() or score(Collector).
Query	weight	The method Query.weight() was left in Query for backwards reasons in Lucene 2.9 when we changed Weight class. This method is only to be called on top-level queries - and this is done by IndexSearcher. This method is just a utility method, that has nothing to do with the query itsself (it just combines the createWeight method and calls the normalization afterwards). For 3.3 I will make Query.weight() simply delegate to IndexSearcher's replacement method with a big deprecation warning, so user sees this. In IndexSearcher itsself the method will be protected to only be called by itsself or subclasses of IndexSearcher.

highly desirable. Last but not least, the generalization of the heuristics calibration cannot be guaranteed by our evaluation.

IV. QUALITATIVE ANALYSIS

This section provides a qualitative analysis of some exemplar paragraphs, identified during the manual validation. The aim is to: (i) show examples of the various kinds of descriptions that the approach is able to mine; (ii) explain why the approach, in some cases, detected false positives; and (iii) explain why the approach missed some good descriptions, i.e., false negatives. In summary, starting from what we collected during our validation, it is possible to classify the retrieved paragraphs as follows:

- *True positive paragraphs:* these are paragraphs identified by the proposed approach, that the human validation classifies as properly describing a given method. Such paragraphs can be used to help understanding the source code or to re-document it.
- *False positive paragraphs:* these are paragraphs identified by the proposed approach, however, based on the human validation, they do not really have the purpose of providing a method description. Such paragraphs reduce the precision of the approach.
- *True negative paragraphs:* these paragraphs are not selected by the proposed approach and, indeed, they do not describe methods, while they possibly refer to a method for other purposes.
- *False negative paragraphs:* these paragraphs are discarded by the proposed approach, however they represent good method descriptions.

The examples reveal several discourse patterns that characterize true positive, false positive, and false negative method descriptions. Regarding *true positives*, these paragraphs are always composed of sentences in *affirmative form*, directly explaining a method's syntax or behavior. For Lucene (Table IV), the first true positive example is a clear description of the *addAttributeImpl* method from the *AttributeSource* class. In this case, the developer initially informs others about the introduction of a new method and after that he explains what the method does: "finds all interfaces that the class or superclasses implement and that extend the Attribute interface" and "adds the interface or instance mappings to the attribute map for each

TABLE V
EXAMPLES OF FALSE POSITIVE PARAGRAPHS FOR ECLIPSE.

Class	Method	Paragraph
Table	releaseWidget	Similar (and related) NPE is on Table class, on releaseWidget() method call - the last element in columns[] array is null.
WorkbenchPart	dispose	It must be the last method called on the contribution item. After calling dispose(), it is a bug to continue using the contribution item
OperationCanceledException	OperationCanceledException	On thinking about it, throwing OperationCanceledException would be unusual since the method does not take a progress monitor parameter. Returning a CANCEL status seems like the best approach.

TABLE VI
EXAMPLES OF FALSE POSITIVE PARAGRAPHS FOR LUCENE.

Class	Method	Paragraph
MultiReader	isOptimized	These 3 methods should probably be fixed: isOptimized() would fail - similar to isCurrent() setNorm(int, String, float) would fail too, similar reason. directory() would not fail, but fall to return the directory of reader[0]. This is because MultiReader() (constructor) calls super with reader[0] again. I am not sure.
SegmentReader	termDocs	Yes, but this class is package private and unused! AllTermDocs is used by SegmentReader to support termDocs(null), but not AllDocsEnum. The matchAllDocs was just an example, there are more use cases, e.g. a TermsFilter (that is the non-scoring TermQuery variant): Just use the DocsEnum of this term as the DicIdSetIterator.
TopDocsCollector	topDocs	We might also consider deprecating the topDocs() methods that take in parameters and think about how the paging collector might be integrated at a lower level in the other collectors, such that one doesn't even have to think about calling a diff. collector

of the found interfaces". This paragraph was extracted from a list of *candidate descriptions* with highest score (cosine=0.74), where each of these paragraph refer to 100% of the method parameters, in this case *addAttributeImpl*. We can find only phrases in *affirmative form* without sentences in *dubitative form*. In same way, for Eclipse (Table III) if we observe the first true positive example—describing the constructor *ServiceLoader*—we can find a paragraph in *affirmative form* without sentences in *dubitative form*. It is important to note that this paragraph, with respect to the first example paragraph of Lucene, obtained highest rank because if refers to 100% of the parameters of *ServiceLoader* and it contains the keywords "call" and "return" and thus it describes the method in terms of invocation of other methods and in terms of its returned value (syntactic descriptions).

If we look at false positives, for Eclipse (Table V) we can notice examples of descriptions that are too specific (e.g., for the *releaseWidget* method), hence not particularly useful to properly understand the entire method. Other examples are related to faulty behavior (*dispose*) and about a possible bug fixing (constructor of *OperationCanceledException*). For Lucene (Table VI), the candidate description of the *isOptimized* method from the *MultiReader* class consists, actually, in a proposal of bug fixing for several methods. Regarding the *termDocs* method from the *SegmentReader* class, the paragraph mainly describes dependencies among methods rather than describing method behavior. In some sense, this could also be considered a true positive (e.g., useful to understand method dependencies), although our evaluators classified it as a false positive because the paragraph did not clearly describe the method behavior. The last case (the *topDocs* method from the *TopDocsCollector* class) is a paragraph where people suggest to deprecate such a method and integrate the behavior elsewhere. Also in this case, the description could be, in principle, considered a useful one, although it was not considered as such because the paragraph described the behavior to be refactored. In conclusions, false positives either concern borderline cases—which could be useful in some cases and hence increase the amount of useful material a developer has to comprehend the source code—or cases such as faulty or future behavior which would not easy to discern automatically. This also suggests that the results strongly depend on the data source we use (i.e., the content of the emails and, above all, of the bug reports), indicating that some sources, such as bug reports, in some case contain descriptions that are not appropriate for describing the correct, current behavior of a method.

Finally, concerning the false negatives (Tables VII and VIII for Eclipse and Lucene respectively), many of them were descriptions discarded because they describe the methods without containing keywords (such as, "return", "override", "invoke", etc.) we used for filtering. For example, in the case of Lucene, the paragraph referring to the *optimize* method from the *IndexWriter* class contains the sentence "I found that IndexWriter.optimize(int) method does..." containing the class name and method name, yet it does not contain any of the above keywords. A similar situation occurs for the *parse* method from the *TrecFTParser* class. Similar examples can be found in Eclipse, where the *doubleClicked* method from the *JavaStringDoubleClickSelector* class is, again, described properly, yet none of the filtering keywords is mentioned. In conclusion, this suggests that some false negatives could have been avoided by weakening the filtering criteria, however this would also have reduced the precision and hence would have increased the amount of (possibly useless) descriptions a developer has to browse.

TABLE VII
EXAMPLES OF FALSE NEGATIVE PARAGRAPHS FOR ECLIPSE.

Class	Method	Paragraph
JavaStringDoubleClickSelector	doubleClicked	What it does is: - change the behavior of the doubleClicked() methods to also consider the endpoint of the mouse selection for its calculation of the text selection.
Engine	accept	If I understood TypeDescriptor.initialize() method correctly, it is not interested in the method code, so you could use classReader.accept(visitor, ClassReader.SKIP_CODE) to completely skip all methods code from visiting. Same applies to implementation of SearchEngine.getExtraction(..) and TagScanner.Visitor.getMethods(..) methods, where you also can add ClassReader.SKIP_CODE to avoid visiting method code.

TABLE VIII
EXAMPLES OF FALSE NEGATIVE PARAGRAPHS FOR LUCENE.

Class	Method	Paragraph
IndexWriter	optimize	I found that IndexWriter.optimize(int) method does not pick up large segments with a lot of deletes even when most of the docs are deleted. And the existence of such segments affected the query performance significantly. I created an index with 1 million docs, then went over all docs and updated a few thousand at a time. I ran optimize(20) occasionally. What saw were large segments with most of docs deleted. Although these segments did not have valid docs they remained in the directory for a very long time until more segments with comparable or bigger sizes were created.
TrecFTParser	parse	In TrecFTParser.parse(), you can extract the logic which finds the date and title into a common method which receives the strings to look for as parameters (e.g. find(String str, String start, int startlen, String end)).

V. RELATED WORK

Our approach relates to previous work both in its goals and execution.

Previous results that are closest to our work and used in our approach (see Section II) were published by Bacchelli *et al.* [4], [5], [10]. What relates this work to our approach is the use of similar heuristics, as well as the main goal of connecting emails and source code. What differentiates our work is the emphasize on methods (rather than classes) and the specific focus on paragraphs describing the methods. Another work concerned with extracting technical information (such as, source code elements) embedded in emails and other unstructured information [11], uses spell checking tools, yet it is not concerned with identifying relevant parts of the code. Recently Bettenburg *et al.* [12] used an approach relying on clone detection to link emails to source code. While the purpose of our work is different, their approach could potentially be used—as we plan to do in future work—to increase the performances of our approach.

Many techniques for traceability link recovery between software artifacts [13]and many recommendation systems [14] aim at connecting specific source code artifacts to unstructured text documents. Traceability link recovery techniques based on text retrieval techniques [1], [2] are usually used to connect source code and text-based documentation. Two issues differentiate all such approaches to traceability from our work: (i) none of them specifically addresses methods and emails as linking artifacts; and (ii) none of them is concerned with the establishing that the external artifacts (i.e., emails in our case) contain specifically description of methods. Likewise, some recommendations systems, such as, Hipikat [15], are based on the use of text retrieval techniques. As opposed to most traceability techniques, Hipikat handles method level granularity and emails and it can recommend emails related to a method, yet it does not extract the descriptive parts of the emails.

Our approach relies on heuristics that capture discourse rules that developers follow when describing code in their communications. Previous work [16], [17] looked into how developer describe problems in bug reports. Rules of discourse in source comments were also investigated. For example, Etzkorn *et al.* [18] found that 75% of sentence-style comments were in the past tense, with 55% being some kind of operational description (e.g., "This routine reads the data.") and 44% having the style of a definition (e.g., "General matrix"). Likewise, Tan *et al.* [19] analyzed comments written in natural language to extract implicit program rules and used these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. In the same realm, Zhong *et al.* [20] proposed an approach, called Doc2Spec, that infers resource specifications from API documentation in natural languages.

One of the potential uses of our approach is the re-documentation of source code by generating method summaries using the paragraphs extracted from the emails and bug reports. Existing work addressed the issues of using the code and comments in methods to generate method summaries. Most recent approaches used language generation techniques [21] and information retrieval techniques [22] to generate method summaries. Some of these summaries are not unlike some of the paragraphs our approach retrieves.

VI. CONCLUSION AND FUTURE WORK

We verified in this work our hypothesis that developer communications, such as, mailing lists and bug reports, contain textual information that can be extracted automatically and used to describe methods from Java source code. We found

that at least 22% of the methods in Eclipse and 65% of the methods in Lucene are specifically referenced in emails and bug reports. Only a part of these references are included in paragraphs that describe the methods and can be automatically retrieved by our approach. Our approach to mine method descriptions from developers communication—and specifically from mailing lists and bug tracking systems, first traces emails/bug reports to classes, and then, after extracting paragraphs, traces them to methods. After that, it relies on a set of heuristics to extract different kinds of descriptions, namely: (i) descriptions explaining methods in terms of their parameters and return values; (ii) descriptions explaining how a method overloads/overrides another method; and (iii) descriptions of how a method works by invoking other methods. Finally, a further pruning is performed by computing the textual similarity between the paragraphs and the method body.

Our empirical evaluation indicates that the proposed approach is able to identify descriptions with a precision up to 79% for Eclipse and up to 87% for Lucene. The method coverage of these descriptions is low for Eclipse, ranging between 7% and 2%, and higher for Lucene, ranging between 36% and 15%. The low method coverage is the result of two factors: (i) only part of the methods are described properly in these communications, and (ii) our approach is rather conservative as we focused on achieving high precision, given the envision usage scenario (i.e., a developer trying to understand quickly what a method does). Our investigation revealed the presence of linguistic patterns that—at least for the two analyzed systems—characterize different kinds of method descriptions.

There are several directions for future work. First, this is a first approach aimed at mining method descriptions from external unstructured artifacts, therefore there is still a lot of space for improvement with the aim of increasing the precision while keeping the method coverage as high as possible, as well as reducing the percentage of false positives. Furthermore, we would aim at further validating the proposed approach on a larger data set, including mailing lists and bug reports from more systems. It is important to establish whether the values we used for the heuristics of our approach work equally well on other data sets. Finally, we would also investigate approaches for mining descriptions of software artifacts at a higher level of abstraction, such as classes and packages.

ACKNOWLEDGMENTS

We would like to thank Annibale Panichella for his help in the manual validation of the results. Andrian Marcus was supported in part by grants from the US National Science Foundation: CCF-1017263 and CCF-0845706.

REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[2] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of 25th International Conference on Software Engineering*. Portland, Oregon, USA: IEEE CS Press, 2003, pp. 125–135.

[3] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*. IEEE Computer Society, 2009, pp. 205–214.

[4] 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*. ACM, 2010, pp. 375–384.

[5] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*. IEEE Computer Society, 2010, pp. 24–33.

[6] D. Klein and C. D. Manning, "Fast exact inference with a factored model for natural language parsing," in *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*. MIT Press, 2002, pp. 3–10.

[7] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-based lightweight C++ fact extractor," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 134–143.

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

[9] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[10] A. Bacchelli, M. Lanza, and V. Humpa, "RTFM (read the factual mails) - augmenting program comprehension with remail," in *Proceedings of the European Conference on Software Maintenance and Reenginering*, 2011, pp. 15–24.

[11] N. Bettenburg, B. Adams, A. E. Hassan, and M. Smidt, "A lightweight approach to uncover technical artifacts in unstructured data," in *Proceedings of the IEEE International Conference on Program Comprehension*, 2011, pp. 185–188.

[12] N. Bettenburg, S. W. Thomas, and A. E. Hassan, "Using fuzzy code search to link code fragments in discussions to source code," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*. IEEE, 2012, pp. 319–328.

[13] J. Cleland-Huang, O. Gotel, and A. E. Zisman, *Software and Systems Traceability*. Springer, February 2012.

[14] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, July/August 2010.

[15] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, 2005.

[16] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 127–134.

[17] A. J. Ko and P. K. Chilana, "Design, discussion, and dissent in open bug reports," in *iConference*, 2011, pp. 106–113.

[18] L. H. Etzkorn, L. L. Bowen, and C. G. Davis, "An approach to program understanding by natural language understanding," *Natural Language Engineering*, vol. 5, pp. 1–18, 1999.

[19] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or bad comments? */," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[20] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring specifications for resources from natural language API documentation," *Automated Software Engineering Journal*, 2011.

[21] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 43–52.

[22] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the IEEE International Working Conference on Reverse Engineering*, 2010, pp. 35–44.