Leveraging Informal Documentation to Summarize Classes and Methods in Context

Latifa Guerrouj, David Bourque, and Peter C. Rigby
Department of Software Engineering
Concordia University, Montréal, QC, Canada
Email: latifa.guerrouj@polymtl.ca, david.bourque@encs.concordia.ca, peter.rigby@concordia.ca

***Abstract*—Critical information related to a software developer's current task is trapped in technical developer discussions, bug reports, code reviews, and other software artefacts. Much of this information pertains to the proper use of code elements (*e.g.*, methods and classes) that capture vital problem domain knowledge. To understand the purpose of these code elements, software developers must either access documentation and online posts and understand the source code or peruse a substantial amount of text. In this paper, we use the context that surrounds code elements in StackOverflow posts to summarize the use and purpose of code elements. To provide focus to our investigation, we consider the generation of summaries for library identifiers discussed in StackOverflow. Our automatic summarization approach was evaluated on a sample of 100 randomly-selected library identifiers with respect to a benchmark of summaries provided by two annotators. The results show that the approach attains an R-precision of 54%, which is appropriate given the diverse ways in which code elements can be used.**

I. Introduction

During the evolution of large software systems, developers must understand a large number of code elements (*e.g.*, *Activity.dispatchTouchEvent*). To understand these code elements, developers turn to documentation, StackOverflow, webseraches, etc. Each of these resources describe the code element in various contexts. For example, if a software task requires an understanding of the Java library *DrmStore* class of the Android package *android.drm*, source code or sentences from documentation about that class will need to be read. Reading the source code or text composing code elements to understand their purpose can be time consuming and frustrating for the developer given the vast quantity of information. One way to reduce the time a developer spends on understanding the propose of code elements trapped in developers discussions is to provide a summary of the purpose of each code element. An accurate summary can enable a developer to reduce the time spent perusing software artifacts pertinent to his/her task.

Recent research has deployed efforts towards automatic generation of natural language descriptions of software. In particular, Haiduc *et al.* have leveraged the lexical and structural information in the source code to summarize source code classes [8]. Moreno *et al.* have generated Java classes' summaries by focusing on the content and responsibilities of the classes while Sridhara *et al.* provides natural language summaries of Java methods based on identifiers' names [21]. The novelty of our approach is to summarize identifiers (*i.e.*, classes and methods) by exploiting identifier context within informal documentation, *i.e.*, information which surrounds the classes or methods being summarized including source code textual information contained in documentation. An identifier context is important for programmers to know because it helps answer questions about why an identifier exists and the rationale behind its usage in the software [6].

The official documentation provides a narrow understanding of the contexts in which a method or class can be used. Our future work is to help provide the summaries of these contexts and redocument the Android API "in context". In such a way developers will have access to concise summaries that describe code elements, their purpose, the rationale behind their usage as well as their dependencies with other code elements that are frequently discussed with them. This summarization approach will be integrated within Integrated Development Environment such as Eclipse to help developers quickly understand the code elements pertinent to their tasks. We plan to further validate our approach using 1) a ROUGE benchmark which is a common evaluation procedure for natural language summarization and 2) user studies which will involve professional developers who will perform programming and maintenance tasks that will require them to understand and modify code elements.

II. Related Work

A. Automatic Natural Language Summarization

Automatic summarization of natural language documents has received a significant attention recently. Researchers such as Rastkar *et al.* [18] and Moreno *et al.* [15] have recently proposed techniques to summarize documentation. Other research works have focused on the summarization of source code and its elements. Murphy [16] has dealt with structural summarization of source code, she has proposed two techniques to support the summarization approach. The first technique, the software reflexion model technique, enables an engineer to summarize selected structural information in the context of a task-specific high-level model. The second technique, the lexical source model extraction technique, supports the summarization process by facilitating the scanning and analysis of system artifacts for structural information that is difficult or impossible to extract at low cost using existing approaches. Recent relevant works on the summarization of code elements are as follows. Haiduc *et al.* generated automatic natural language summaries of Java classes by leveraging the lexical and structural information. They used techniques based on the

position of terms in the source document and others relying on text retrieval [8]. Moreno *et al.* generated Java class summaries by focusing on the content and responsibilities of the classes, rather than their relationships with other classes [15]. Sridhara *et al.* provides natural language summaries of Java methods. Their approach selects a method's most important statements and then extracts keywords from the identifier names in those statements. A natural language generator is then used to stitch the keywords into English sentences to form a method summary [21]. A more recent contribution to this area generates automatic documentation via the source code summarization of method's context. This technique includes context by analyzing how the Java methods are invoked [13].

B. Code Element Extraction in Documentation

Information retrieval based Traceability techniques have been used to resolve the links between source code elements and documentation. For example, Antoniol *et al.* used probabilistic and Vector Space Model (VSM) to resolve terms [1], while Marucs *et al.* use Latent Semantic Indexing (LSI) [12]. The techniques using LSI attain an average precision of 0.42 and recall of 0.38 while the VSM-based approaches show an average precision of 0.23 and recall of 0.31.

Bacchelli *et al.* have suggested an approach, called Miler, that relies on programming language coding standards in the form of lightweight regular expressions to extract code elements contained in email discussions. Miler's average precision and recall is 0.33 and 0.64 respectively [2].

Recently, Rigby and Robillard have suggested an approach called ACE that uses an island parser to identify code elements in documents [19]. Subramanian *et al.* only extracts code elements from code snippets [22]. In contrast, ACE extracts code elements from freeform text and code snippets that do not necessarily compile with an average precision and recall at or above 90%. The output of the ACE is a list of the code elements associated with each each documents, *i.e.*, Android StackOverflow posts in our case [19].

III. Summarizing Library Identifiers

A. Stage 1: Extracting library identifiers from free-form text

We used the approach suggested by Rigby and Robillard, ACE, to extract library identifiers that exist in the free-form text, *i.e.*, StackOverflow posts in our study. ACE identifies not only code elements present in free-form text but it also handles code fragments that may not be compilable as well as large document collections with high precision and recall [19]. It uses an island parser to identify code elements in documents. Unlike prior works, this approach does not depend on an index of valid elements parsed from the source code of a particular system. Instead, it identifies code elements in Java constructs and creates an index of valid elements based on the elements contained in the collection of documents [19].

B. Stage 2: Creating language models for library identifiers

Documentation pre-processing: We preprocessed the collection of documents to normalize their text. This step includes removing html tags, stop words, operators, programming language keywords, and special characters as well as common English words. Additionally, compound identifiers that were part of an identifier context were split using the facto CamelCase splitter widely applied in traceability recovery and feature location tasks for Java programs (*e.g.*, [7]). We consider only posts that are valid answers from the StackOverflow to avoid having summaries that misguide developers since the StackOverflow may also contain erroneous answers. Valid answers are those having high votes by developers in the StackOverflow. Also, for each post, we removed, using appropriate regular expressions, the code snippets that are part of a discussion but not the source code comments since they convey relevant information about code entities [7].

Library identifier's context: We exploit the *local context* of identifiers to provide accurate summaries of their purpose. As in Dagenais and Robillard's work [5], the local context of an identifier consists of all the terms co-occurring with it in the same document (*i.e.*, pre-processed StackOverflow discussion). We also consider the *term proximity*. In the case of local context, proximity is the distance in characters between an identifier and n terms immediately surrounding it. In our case, we consider the most closest terms surrounding identifiers. Thus, our context is limited. We consider a limited local context because we assume the information surrounding and closer the code elements is more likely to be relevant to them than the one further apart or in a different context. We experimentally determined the length of the context whose value is reported in Section V.

Library identifier's language models: We create a language model for each identifier found in our documentation by exploiting its context. The co-occurrences of words from the identifier context with the identifier in question is able, in effect, to capture the usage, purpose, and rational behind each identifier. An n-gram language model is a sequence of n tokens that appear consecutively in the text. To create an n-gram language model, we calculate the frequency of all 1-grams, 2-grams, up to n-grams. A 1-gram is the frequency of all the distinct terms in a corpus, a 2-gram counts the number of pairs of terms that co-locate, and so on. A n-gram model assumes a Markov property that the occurrence of the next word in an n-gram is dependent only on the preceding n-1 words [4]. We use simple n-gram language models because they are sufficient to show that, like natural language, code follows regular and predictable patterns [9]. The predictable nature of software development has been exploited to generate automatically programming examples [3], auto-complete sections of code [19], and automatically translate between Java and C# [17].

C. Stage 3: Selecting words using an unsupervised learning

Selecting word features: We consider simple 1-grams in this preliminary study to include in the identifier summaries. Yet, our work-in-progress is investigating the use of n-grams with $n \geq 2$. We measure the relevance of 1-grams with respect to the collection of analyzed documents based on the $tf-idf$ measure which is widely used in practice for document

retrieval tasks. The $tf - idf$ numerical statistic reflects how important a word is to a document in a collection or corpus [20] which consists in our case on all identifers' contexts instead of the entire collection of original posts.

Selecting words using an unsupervised learning: We used the K-means clustering to select words to include in an identifier summary. This clustering algorithm finds groups of words in order to achieve, in the one hand, the highest possible similarity between words of a group, in the other hand, the highest possible dissimilarity between words of different groups. K-means represents each sentence in a vector space model [14]. So, each identifier context is represented as a vector of features, where the features correspond to the different words in the identifier context. For measuring the similarity between two sentences the Euclidean distance is used defined as: $Distance(X, Y) = \sum_{i=1}^{n}(x_i - y_j)^2$ where X and Y are sentences represented as vectors with n features. The above two steps hold for any size of n-grams.

IV. Data Source and Benchmark

A. Data source

StackOverflow is a question and answer forum for professional developers [10][1]. Developers ask and answer questions as well as provide their votes on the quality of a post. Each post is related to a specific topic that often discusses issues involving code elements. We processed all question and answer posts of StackOverflow tagged between August 2008 and January 2014. We focus on the Android project in this investigation. Android has become the dominate mobile development platform with 71% of practitioners[2]. This popularity makes Android interesting for study and analysis. The goal from this preliminary investigation is to validate whether we would be able to summarize Android library identifiers using their context which we experimentally determined as 100 unique 1-grams, 50 words left from the identifier and 50 others right from it. The data gathered allowed us to have a database of 2.4 millions StackOverflow posts of the Android project having a total of 23 million library identifiers where 16 million are types and 7 million represent methods. We randomly-sampled from this initial set, a sample of 100 Java library identifiers belonging to the package **Android.*** which consists of 50 classes and 50 methods to be summarized. Our data is available for replication up on request.

B. Benchmark of summaries by human annotators

We evaluated our approach against a benchmark of a manually-built summaries. We created short manual descriptions for the 100 analyzed Android library identifiers using the online Android Java documentation[3] plus other sources of information including source code, comments, etc. to avoid any possible bias. The manual identifier summaries consists of a set of relevant contextual words that describe the purpose of an identifier. The number of key words included in the summaries varies depending on the length and complexity of identifiers. During the benchmark building process, the two human annotators followed a consensus approach, *i.e.*, one author proposed a summarization, which was then verified and validated by a second author. In a few cases, disagreements were discussed among all the authors. We adapted this approach in order to ensure the interrater reliability of our study and minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers' domain and solution knowledge, experience, personal preference, etc.

[1] *http://developer.android.com/*

[2] *http://www.developereconomics.com/reports/q3-2013/*

[3] *http://developer.android.com/*

V. Evaluation of the Library Identifiers Summaries

How accurate are the automatic generated summaries with respect to the summaries created by the human annotators?

To evaluate how well an automatic generated summary resembles the oracle summary in the manual benchmark, we computed the R-precision, an evaluation metric, widely used the field of information retrieval [11]. R-precision is computed in a similar way to precision at-k, the precision for summaries of length k. Precision-at-k determines out of the top k relevant words returned by our automatic approach ($predicted_K$), how many are correct ($|oracle \cap predicted_K|$). We use R-precision because it has the advantage of being able to handle summaries of variable lengths. The R-precision evaluates the top R relevant words returned by our automatic approach (R) $predicted_R$ where R is the length of the summary oracle. More formally, R-precision is given by:

$$R\text{-precision} = \frac{|oracle \cap predicted_R|}{|predicted_R|}$$

VI. Study Results and Discussion

This section reports the results of the empirical evaluation preformed on the 100 randomly-sampled Android identifiers. Table I reports descriptive statistics (1st quartile, median, 3rd quartile, mean, max) of the accuracy of our novel approach in terms of R-precision for the analyzed classes. As indicated in Table I, the approach attains, on average, 54% in terms of R-precision.

TABLE I
R-Precision of Automatic Generated Summaries for Classes.

Metric	1Q	Median	Mean	3Q	Max
R-Precision	0.3333	0.5000	0.5443	0.6905	1.0000

Table II reports the same descriptive statistics of the R-precision when dealing with methods. It shows that the approach attains 54% in terms of R-precision when summarizing methods. These quantitative results reveal the ability of our approach to provide summaries for library identifiers with a reasonable average of R-precision. Given the wide range of contexts in which an identifier can be used, it is difficult to accurately identify concise summaries in a manual coding

for all identifiers which may had impacted the obtained R-precision. Involving professional Android developers as annotators in the manual-building process of the summaries' benchmark is important to provide a precise idea about the approach's accuracy. We are also working on the improvement of our results by exploiting other sources of information that discuss library identifiers including bug reports and developers' discussions which may be used as an alternative when no contextual information is provided from the StackOverflow.

TABLE II
R-PRECISION OF AUTOMATIC GENERATED SUMMARIES FOR METHODS.

Metric	1Q	Median	Mean	3Q	Max
R-Precision	0.3500	0.5000	0.5373	0.7292	1.0000

Inaccurate summaries provided by the suggested approach were mainly due to the lack of sufficient identifier's context. For example, we faced some discussions where only code snippets and examples were provided by developers. Such code examples are seldom commented. We believe that it is impossible to provide an accurate summary for an identifier without a minimum language text description that discusses it in the context of a specific problem at hand. We also believe that even if we consider the source code comments found in code examples (if any), these comments should be meaningful to fully benefit from them in the summarization task. Finally, inaccurate summaries were also due to code elements that contain short terms as well as abbreviations (*e.g.*, class *DrmStore*) and that were part of an identifier context. Such complex identifiers need advanced identifier splitting and expansion techniques such as [7] that we are investigating in our work-in-progress.

VII. CONCLUSION AND FUTURE WORK

To help developers quickly understand library identifers discussed in informal documentation, we suggest a novel approach that generates automatic identifier summaries using their context. The novelty of our approach is to exploit informal documentation including source code textual information embedded in it to provide accurate summaries. Its evaluation on a randomly-chosen sample of 100 Android identifiers from StackOverflow has shown an R-precision of 54% with respect to the summaries generated by human annotators. As future work, we plan to experiment our approach with different sizes of n-grams, leverage other sources of information such as developers' discussions and bug reports, and conduct controlled user studies with professional developers to show its relevance in the context of concrete tasks. Unlike the official documentation that discusses code elements in a limited context, our summaries include a wide breadth of contexts that should help developers quickly understand code elements pertinent to their tasks.

REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *32nd ACM/IEEE International Conference on Software Engineering*, pages 375–384, 2010.

[3] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 782–792, 2012.

[4] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *ACL*, pages 310–318, 1996.

[5] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *34th ACM/IEEE International Conference on Software Engineering*, pages 47–57, 2012.

[6] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar apis: An exploratory study. In *ICSE*, pages 266–276. IEEE, 2012.

[7] L. Guerrouj, D. P. Massimiliano, G. Yann-Gaël, and G. Antoniol. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, pages 575–599, 2013.

[8] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 223–226, 2010.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 35th International Conference on Software Engineering*, pages 837–847, 2012.

[10] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest q&a site in the west. In *CHI*, pages 2857–2866, 2011.

[11] C. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*. Addison-Wesley Publishing Company, University Press, 2008.

[12] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.

[13] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 279–290, 2014.

[14] J. McQueen. Some methods for classification and analysis of multivariate observations. In *Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[15] L. Moreno, J. Aponte, G. Sridhara, M. A., L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC'13)*, pages 23–32, 2013.

[16] G. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. University of Washington.

[17] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 651–654, 2013.

[18] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Trans. Software Eng*, 40(4):366–380, 2014.

[19] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *32nd ACM/IEEE International Conference on Software Engineering*, pages 832–841, 2013.

[20] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24:513–523, 1988.

[21] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 43–52. ACM, 2010.

[22] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *International Conference on Software Engineering*, pages 563–572, 2014.