Searching Stack Overflow for API-usage-related Bug Fixes Using Snippet-based Queries

Eduardo C. Campos
Faculty of Computing
Federal University of
Uberlândia, Brazil
eccampos@ufu.br

Martin Monperrus
University of Lille & Inria
Lille, France
martin.monperrus@univ-lille1.fr

Marcelo A. Maia
Faculty of Computing
Federal University of
Uberlândia, Brazil
marcelo.maia@ufu.br

ABSTRACT

Project-specific bugs are related to the misunderstanding or incomplete implementation of functional requirements. API-usage-related bugs are independent of the functional requirements. Usually they cause an unexpected and wrong output or behavior because of an incorrect usage of an API (Application Programming Interface). We propose an approach to find fixes for API-usage-related bugs, which is based on matching snippets being debugged against related snippets in a Q&A website (Stack Overflow). We analyzed real code excerpts from OHLOH Code Search containing API method calls that may lead to API-usage-related bugs depending on how they are used by developers. We conducted a study with these excerpts to verify to what extent the proposed approach provides proper information from Stack Overflow to fix these potential API-usage-related bugs. The results are encouraging: 66.67% of Java excerpts with potential API-usage-related bugs had their fixes found in the top-10 query results. Considering JavaScript excerpts, fixes were found in the top-10 results for 40% of them. These results indicate that our approaches (i.e., *lsab-java* and *lsab-js* combined with the keyword filter) outperform Google and Stack Overflow in searching for API-usage-related bug fixes.

CCS Concepts

•Software and its engineering → Software libraries and repositories;

Keywords

API-usage-related bugs, crowd debugging, crowd knowledge

1. INTRODUCTION

Developers frequently need to use methods they are not familiar with or they do not remember how to use [16]. Unfortunately, some of the most severe obstacles faced by developers learning a new API are related to its documentation, in particular because of scarce information about API's design, usage scenarios, and code examples [3][21]. As a result, developers may write code inconsistent with API documentation and thus introduce bugs [33].

The battle against software bugs exists since software existed. It requires much effort to fix bugs, e.g., Kim and Whitehead [8] report that the median time for fixing a single bug is about 200 days. The process of debugging is strongly related to the nature of the bug. For example, debugging a segmentation fault means focusing on allocation and deallocation of memory, while debugging an error of *NullPointerException* means finding which line of code is accessing some attribute or operation (API element) on a null variable.

In this paper, we claim that there exists a class of bugs that relates to the common misunderstanding of an API. Those bugs have a common characteristic: they occur repeatedly in different contexts and they are independent of the application domain. For instance, there are plenty of JavaScript programmers who experienced that *parseInt("08")* returns 0 whereas the expected result is 8. This class of bugs is called "API-usage-related bugs", because it is likely that this kind of bug has already occurred several times and there exists a description of the problem somewhere on the web, along with its explanation and fix. In other terms, the crowd has already identified the bug and its solution.

Generally speaking on debugging, one common limitation of research techniques that support automating or semi-automating debugging is their reliance on a set of strong assumptions on how developers behave when debugging (e.g., the approaches tend to assume perfect bug understanding) [18]. A more flexible solution would allow developers to inform a suspicious code snippet and get extra knowledge that would help understanding what would be possibly wrong with that pattern of code. In fact a common behavior of developers is to post pieces of code that contain bugs in question-answer (Q&A) sites, in order to obtain a diagnosis of correction for that particular bug. Treude et al. [28] pointed out that Stack Overflow (SO) is particularly effective for code reviews, for conceptual questions and for novices.

So, for programmers who experience an API-usage-related bug, the SO crowd can be asked for a way to fix the bug using their own suspicious snippets. However, such approach would require an effective way to search the crowd knowledge using suspicious code snippets.

General purpose search engines were not designed to receive snippets as queries: typically these engines do not work well with large queries. Alternatively, different code mining techniques and tools have been proposed to retrieve rele-

CASCON '16 Oct. 31 – Nov. 2, 2016, Toronto, Canada

DOI: 10.1145/1235

vant software components from different repositories with the aim of assisting developers during the software reuse process (e.g., Sourceforge [1], Google Code Search[2], Exemplar [13]). However, these customized code search engines were designed to index all information of software projects (e.g., help documentation, API calls used inside each application, textual descriptions of applications, etc.) instead of only code snippets. Moreover, code search engines were not designed to allow snippets being used as query. For instance, Google Code Search allows regular expressions and typical fields, such as, file/class/method names. Other engines like, Sourceforge and Exemplar allow keywords as input. In other words, these kind of engines are not suitable to match developers' snippets with crowd's snippets.

Therefore, our approach aims to recommend posts of SO whose code match developers' snippets. The motivation is that the answer's text provided by SO community members along with the code snippets could be helpful to assist developers during the debugging tasks at hand because they show how to use these code snippets, i.e., the whole post may help to overcome the cognitive distance [10] to understand problems in these code snippets.

To sum up, our contributions are as follows:

- An empirical analysis of code-snippet matching using code snippets of SO written in Java and JavaScript programming languages. This analysis showed that the built-in search engine of SO is not adequate when used with code snippets as input query;
- The construction of specific indexes for Apache Solr[3] containing lexical and syntactic information of source code snippets written in Java and JavaScript programming languages. Those indexes serve as core infrastructure for a recommendation system;
- The construction of a dataset containing 30 code excerpts (i.e., snippets) with potential API-usage-related bugs picked from OHLOH Code Search [4] (i.e., 15 excerpts in Java and 15 in JavaScript) and 30 posts of SO that fix the respective potential API-usage-related bug;
- Our recommendation strategies for code excerpts with potential API-usage-related bugs (i.e., *lsab-java* and *lsab-js* combined with the keyword filter) outperform Google and SO, and 66.67% of Java excerpts and 40% of JavaScript excerpts could have a proper recommendation in the top-10 results.

The remainder of this paper is organized as follows. Section 2 presents the concept of API-usage-related bugs and an empirical study of them. Section 3 presents the methodology of our work. Section 4 presents the experiments performed with code snippets of SO. Section 5 shows the experiment conducted with code excerpts that contain potential API-usage-related bugs. Related work is surveyed and shown in Section 6. In Section 7, we draw our conclusions.

[1]http://sourceforge.net/ (verified 05/02/2016)
[2]http://www.google.com/codesearch (verified 05/02/2016)
[3]http://lucene.apache.org/solr/ (verified 05/02/2016)
[4]https://code.openhub.net/ (verified 05/02/2016)

2. API-USAGE-RELATED BUGS

API-usage-related bugs are not bugs in functions or methods of an API. In this paper, we suppose that the API does not contain bugs. API-usage-related bugs occur due to an incorrect usage of the functions or API methods by the developer. In opposition to API-usage-related bugs, project-specific bugs relate to misunderstanding or incorrect implementation of domain concepts.

Definition: An "API-usage-related bug" is a bug that causes an unexpected and wrong output or behavior resulting from an incorrect usage of an API.

Let us illustrate the definition with some examples. In Java, there is a method of "java.lang.Math" library called **cos**. The full signature of this method is "public static double cos(double)". The argument of this method is an angle, in radians. This method returns the trigonometric cosine of an angle. Despite this apparently simple description and self-described name, this method poses problems to many developers, as witnessed by the dozens of Q&As on this topic on Q&A websites [5]. Many Q&As relate to the same issue: *Why does Math.cos() give a wrong result?*. One of the reasons is that if the argument of **Math.cos** is passed in degrees rather than radians, the method gives wrong result. Why is the question asked again and again? We hypothesize that the semantics of **Math.cos** is counter-intuitive for many people (they erroneously assume that the angle must be passed in degrees), and consequently, the very same issue occurs in many development situations, independently of the domain. The fix for this problem is to use the method **Math.toRadians** to convert an angle to radians, i.e., "Math.cos(Math.toRadians(angle))".

Let's consider another example in JavaScript. There is a function called **parseInt**, which parses a string given as input and returns the corresponding integer value. This function also poses problems to many developers, as witnessed by the dozens of Q&As on this topic on Q&A websites [6]. Again, many Q&As relate to the same issue: *Why does parseInt("08") produce 0 and not 8?*. The answer is that if the argument of **parseInt** begins with 0, it is parsed as octal (base 8). The fix for this problem is to specify the base. We hypothesize that the semantics of **parseInt** is counter-intuitive for many people (they erroneously assume that the base is 10 rather than 8, perhaps because of the numerical decimal system is most frequently used in general operations involving numbers).

How to fix API-usage-related bugs? To fix API-usage-related bugs, we propose a system that relies on code snippet matching between the suspicious snippet under debug and the related pieces of code present in the SO questions. When facing an API-usage-related bug, the developer would ask our system to be pointed to relevant answer on SO. Our system is based on the fact that Q&As of SO often contain API-usage-related bug fixes. To sum up, *to fix an API-usage-related bug, ask the crowd using your suspicious snippet* [15].

Why does our approach use code snippets as input query instead of terms? We claim that searching for bug fixes using snippets instead of terms is more suitable to fix

[5]http://stackoverflow.com/search?q=Math.cos+java
[6]http://stackoverflow.com/search?q=%5Bjavascript%5D+title%3AparseInt

API-usage-related bugs because understanding the root cause of a failure for developers typically involves complex activities such as navigating program dependencies and rerunning the program with different inputs [18]. Moreover, API-usage-related bugs do not have exception stack traces that allow to locate the counter-intuitive API method.

Where do API-usage-related bugs come from? They appear when a developer of an API makes design decisions that go against the common sense, "common sense" being defined as the intuition and expectation being hold by many developers. In the **parseInt** example, most people expect that **parseInt("08")** returns 8. We see at least three main reasons behind API-usage-related bugs:

- First, the API under consideration may be poorly documented [3][31]. For example, the official documentation of the API can not contain code snippets that teach how to use a particular API method [16][24][29];
- Second, the API under consideration may seem comprehensible enough for some developers who *do not read* the documentation;
- Third, the API may assume something *implicitly*. For instance, **parseInt** *implicitly* assumes that prefixing the string with "0" means that an octal base is chosen.

A detailed study of why API-usage-related bugs appear is out of scope of this paper, it requires inter-disciplinary work between different fields such as software engineering and psychology.

Beyond the toy examples we have just discussed so far, there should be sufficient API-usage-related bug fixes on SO to motivate a recommendation system for this kind of bug. However, it is not possible to manually assess whether the millions of posts of SO refer to API-usage-related bugs or not. Hence, we need an automated technique to assess approximately the space of "API-usage-related bug fixes". This estimate has the only goal to know if there are sufficient occurrences of these kind of fixes to motivate a recommendation system.

We adopted the following criteria to identify SO threads (question + answers) as potentially containing an "API-usage-related bug fix". First, it must contain one snippet in the question since our proposed approach consists of understanding the piece of code queried in our recommendation system. Second, they should be limited to a particular language. Third, they should have an accepted answer to be sure that the crowd was able to solve the bug. In order to select Java or JavaScript API-usage-related bugs, we use a filter on the tagging metadata of the question (i.e., we select questions if and only if they are tagged by "javascript" or "java"). The March 2013 dump of SO contains 3,389,743 posts. By applying the 3 filters aforementioned, we obtain 179,885 and 197,158 posts for Java and JavaScript, respectively. In our opinion, these values are sufficient numbers, *supporting our intuition that there is a wealth of information in API-usage-related bug fixes on SO.*

3. METHODOLOGY

In this section, we present four research questions about code snippet search and API-usage-related bugs ($\mathbf{RQ_1}$, $\mathbf{RQ_2}$, $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$), the programming topics that will be subject of assessment, the preprocessing functions for code snippets and the local SO database used to conduct experiments with SO code snippets (i.e., Experiments I and II).

Before detailing the experiments, we will briefly explain what is indexing. The standard technique of indexing-based search consists of two transformation functions t_{index} and t_{query}. The first function t_{index} transforms the documents (i.e., posts of SO) into terms, and the second function t_{query} transforms the query into terms as well. The ranking consists of matching the document terms against the query terms. The functions t_{index} and t_{query} are not necessarily identical, since the nature of documents (style, structure, etc.) is often different from the nature of queries.

In our proposed approach, the terms are preprocessed code snippets. Moreover, the result of a search on Apache Solr's index is a ranked list of documents (i.e., posts of SO), in which the first one is the more similar to the search query and the last one is the less similar. Each post in this ranking has a numeric value that we call *Solr's score* that represents its similarity to the query (i.e., it is done by Apache Solr in the current implementation). Thus, the first post of the ranking has the greater *Solr's score* and the last one has the smallest value.

3.1 Research Questions

This subsection presents the four research questions considered in the study.

$\mathbf{RQ_1}$: *What is the accuracy of SO query mechanism when input queries are code snippets?*

The research question $\mathbf{RQ_1}$ intends to investigate whether the search engine of SO can cope well with snippet-based queries. **Experiment I** will be conducted to answer this research question. Given a code snippet present in some post of SO and using this same code snippet as input to the built-in search engine of SO, we want to assess how well this search engine can find the post that contains the queried code snippet.

$\mathbf{RQ_2}$: *Which indexing mechanism for code snippets is more effective for the topics considered in this study: the native index of SO site or a customized index using Apache Solr?*

The research question $\mathbf{RQ_2}$ intends to investigate whether the construction of Apache Solr indexes containing information of code snippets improves the code-based queries. We compared the results obtained in the **Experiment I** with the results obtained in the **Experiment II**.

$\mathbf{RQ_3}$: *Which preprocessing function for code snippets best improves the effectiveness of Solr indexes?*

The research question $\mathbf{RQ_3}$ intends to investigate to what extent the use of preprocessing functions for code snippets during the phases of indexing and querying improves the efficacy of code-based query and which preprocessing function provides the best improvement. We conducted the **Experiment II** to answer this research question. In this experiment, we build different indexing alternatives for Apache Solr.

$\mathbf{RQ_4}$: *How effective is the ranking produced by the proposed recommendation system to find fixes for potential API-usage-related bugs?*

The research question $\mathbf{RQ_4}$ intends to investigate how the proposed recommendation system copes with potential

API-usage-related bugs present in the code excerpts of real software projects. We conducted the **Experiment III** to answer this research question. This experiment is dedicated for potential API-usage-related bugs and uses code excerpts picked from real software projects hosted on *OHLOH Code Search* website [7]. These code excerpts contain some API method calls that can lead to API-usage-related bugs. We collected a sample of API method calls on SO site that have this behavior in order to build a dataset with potential API-usage-related bugs.

3.2 Considered Topics

We considered the following topics in our study: **Java Android**, **Java non-Android** and **JavaScript**. We decided to investigate Java Android separately because mobile application development is a trendy topic, with an upward-tendency that increments fast [12]. By also considering plain Java Q&As, we can investigate whether our preprocessing functions for code snippets behave according to specific software platforms (such as Android). The JavaScript related Q&As explore the domain of web development which is among the most active topics on SO.

3.3 Preprocessing functions for code snippets

Code snippets have a different nature compared to natural language texts. So, traditional information retrieval techniques for text preprocessing may not produce the desired terms for index construction. We define some preprocessing functions for code snippets and constructed specific indexes of Apache Solr using these functions in order to perform the Experiments I, II and III. We investigated six preprocessing functions listed below for the Java and JavaScript programming languages.

3.3.1 Function "raw"

This identity function does not modify the content of the piece of source code, i.e., does not perform any code preprocessing.

3.3.2 Term Extraction for Java and JavaScript (pp1)

This function returns all alphanumeric sequences present in the code snippet using the regular expression **"[0-9a-zA-Z_$]+"**. The main desired effect is to remove all punctuation characters.

3.3.3 Syntactic Function for Java (pp2)

This function considers elements of the context-free grammar of the language: it processes the Java code snippet and returns a set of compound terms corresponding to nodes of the abstract syntax tree of this code snippet. They are: type in variable declaration (e.g., **int**), method signature (e.g., **void onAnimateMove[float, float, long]**) and method invocation (e.g., **onAnimateMove**). We use the Eclipse JDT Core Component [8] to generate the abstract syntax tree for the code snippet.

3.3.4 Lexical Function for JavaScript (pp3)

This function returns a list of lexical values (i.e., variable names and string literals) for the JavaScript code snippets using the Rhino API [9].

3.3.5 Lightweight Syntactic Analysis and Binding for Java (lsab-java)

This function processes the Java code snippet and extracts their API method calls. The characteristics of this parser are:

- *Lightweight Syntactic Analysis*: the parser processes the code snippet and identifies all variable declarations and all API method calls occurring in the code snippet (e.g., **Calendar x;** produces the term **Calendar**);
- *Lightweight Binding*: the parser resolves the bindings identified in the previous step, i.e., the syntactic analysis phase (e.g., **Calendar x; x.getInstance**() => the variable access "x" is replaced by the respective type, resulting in term **Calendar.getInstance**()). Moreover, this parser generates as output the terms: **Calendar.getInstance** and **getInstance**. We decided also to index the method name (i.e., the name that appears after the ".") to cover the cases where it is not possible to resolve the binding (i.e., some code snippets of SO do not have the variable declaration statement).

3.3.6 Lightweight Syntactic Analysis and Binding for JavaScript (lsab-js)

This function processes the JavaScript code snippet and extracts their API method calls. It performs the same steps made by the parser *lsab-java*, but contains parsing routines specific to the JavaScript language (e.g., the word "function" is a reserved word of the JavaScript language while this same word is not a reserved word of the Java language). When a developer declares a specific application domain function, the reserved word "function" is used (e.g., **function functionName()**). In this study, we decided not to index the names of application domain functions, since we are interested only in the API method calls. The project-specific texts (e.g., project method name) would not match the bug appearing in the Q&A web site [4].

Table 1 shows the preprocessing functions for code snippets per type (i.e., No preprocessing, Lexical and Syntactic) and programming language (i.e., Java and JavaScript) considered in this study.

Table 1: Preprocessing functions for code snippets per type and programming language.

Type	Java	JavaScript
No preprocessing	raw	raw
Lexical	pp1	pp1, pp3
Syntactic	pp2, lsab-java	lsab-js

3.4 Local SO Database

We downloaded a release of SO public data dump[10] (the version of March 2013) and imported the data into a relational database in order to perform experiments with SO code snippets. The "posts" table of this database stores all questions posted by questioners in the website until the date the dump was built (3,389,743 questions). This table also stores all answers to each question, if any.

[7] https://code.openhub.net/ (verified 05/02/2016)
[8] http://www.eclipse.org/jdt/core/component
[9] https://developer.mozilla.org/en−US/docs/Mozilla/Projects/Rhino
[10] http://blog.stackoverflow.com/category/cc-wiki-dump/

We used the Java library HTML Cleaner[11] in order to extract the code snippets of the questions and answers of SO. We have identified the presence of code snippets in the posts of SO through the use of HTML tags "<pre> <code>...". The next section shows the experiments with SO snippets conducted in this study.

4. EXPERIMENTS WITH CODE SNIPPETS FROM SO

We conducted some experiments with the search engine of SO and the search engine of Apache Solr to answer the research questions (**RQ$_1$**, **RQ$_2$**, and **RQ$_3$**). Subsection 4.1 aims to answer the research question **RQ$_1$** and presents the experiments conducted with SO native site index.

4.1 Experiment I: Querying SO native site index

We investigate whether the search engine of SO is able to cope well with code-based queries. Our idea was to query the search engine of SO using code snippets that already exist in SO. We consider that SO handles code-based queries well if the post that contains such code snippet is rated in the top-10 positions of search engine results. This situation corresponds to the case where the developer queries our recommendation system with a snippet that contains some potential API-usage-related bug and gets the fix for his problem. In cases where SO does not find the posts that fix these potential API-usage-related bugs, we say that this site does not respond well to queries that are code snippets. This is an indication that a special indexing technique needs to be developed for better search results. We conducted the following experimental procedure in the following order:

- **Step 1**: We randomly selected 1000 code snippets from SO for each of the topics considered in this study (i.e., "Java Android", "Java non-Android" and "JavaScript");
- **Step 2**: For each selected code snippet, we save the SO identifier of the post that contains the snippet;
- **Step 3**: We perform two different queries in the SO site for each picked code snippet: a query considering the code snippet in its natural state (i.e., function "raw") and another query considering the result generated by the preprocessing function "pp1" in the code snippet;
- **Step 4**: For each queried code snippet, we calculate in which position in the top-10 the post of SO that contains the code snippet under study (i.e., saved in **Step 2**) appears in the ranking.

Experiment I enabled us to answer the research question **RQ$_1$**.

[11]http://htmlcleaner.sourceforge.net/

Summary of RQ$_1$. The results of Experiment I showed that SO does not have good efficacy when dealing with code-based queries. In addition, this experiment confirms the results obtained by Monperrus et al. [15].

This leads to: 1) preprocessing the code snippets to improve the efficacy of API-usage-related bug matching; 2) build several dedicated indexes containing the information of SO's snippets to investigate whether these indexes would be more suitable for code-based queries, i.e., whether these indexes would yield better results than using the SO native site index. We constructed all these dedicated indexes using the Apache Solr search engine.

Table 2 shows the results obtained in Experiment I.

4.2 Experiment II: Building and Querying the indexes *raw*, *pp1*, *pp2*, *pp3*, *lsab-java* and *lsab-js* of Apache Solr

We built eight Apache Solr indexes to answer research questions **RQ$_2$** and **RQ$_3$**. They are: **Java-raw**, **Java-pp1**, **Java-pp2**, **Java-lsab**, **JavaScript-raw**, **JavaScript-pp1**, **JavaScript-pp3**, and **JavaScript-lsab**. Each index consists of all Java or JavaScript code snippets from local database processed with the respective preprocessing function (e.g., **Java-lsab** index consists of all Java code snippets from local database processed with the preprocessing function *lsab-java*).

As previously stated in Experiment I, for each topic (i.e., Java Android, Java non-Android and JavaScript), we use 1000 code snippets. These snippets were also used to perform the Experiment II. For each Java code snippet, we perform a query on each of the four indexes of Apache Solr (i.e., **Java-raw**, **Java-pp1**, **Java-pp2**, and **Java-lsab**). In the case that the code snippet is written in JavaScript, we perform a query in each of the four remaining indexes of Apache Solr (i.e., **JavaScript-raw**, **JavaScript-pp1**, **JavaScript-pp3** and **JavaScript-lsab**).

For example: given a Java code snippet (i.e., the snippet belongs to the Java Android or Java non-Android topics), it is processed using the following preprocessing functions for Java code: *raw*, *pp1*, *pp2*, and *lsab-java*. After preprocessing the code using one of the preprocessing functions, the generated output is searched in the respective index of Apache Solr. We define the following rule in order to make the index terms (i.e., t_{index}) become as similar as possible to the query terms (i.e., t_{query}): *Code snippets preprocessed with the function X only consult the Apache Solr index that was constructed using the same preprocessing function X.*

Experiment II allowed us to answer the research questions **RQ$_2$** and **RQ$_3$**.

Summary of RQ$_2$. The results of Experiment II show that searching in an index of Apache Solr previously constructed is much more effective than searching in the SO native site index. The column "Not Found" of Table 3 shows that in all indexes, the number of not found results was $\leq 26\%$ (much less than those of Table 2, e.g., for JavaScript topic, we obtained 86% for $t_{index} = SO$ and $t_{query} = pp1$). Moreover, the column Rank ≤ 10 of Table 3 shows the high performance of all indexes in the code-based queries ($\geq 61.8\%$).

Table 2: Using SO Snippets as Code-based Queries (SO refers to Stack Overflow, hence t_{index}=SO means that the indexing function is that of Stack Overflow, "raw" means that no preprocessing is used). "PP" refers to preprocessing.

Topic	Index PP	Query PP	Not Found	Rank #1	Rank ≤ 5	Rank ≤ 10
Java Android	t_{index}=SO	t_{query}=raw	89.3%	8.9%	9.8%	9.9%
Java Android	t_{index}=SO	t_{query}=pp1	94.7%	4.7%	4.9%	5.2%
Java non-Android	t_{index}=SO	t_{query}=raw	90.4%	8.7%	9.4%	9.4%
Java non-Android	t_{index}=SO	t_{query}=pp1	89.1%	10.5%	10.5%	10.6%
JavaScript	t_{index}=SO	t_{query}=raw	95%	4.5%	4.9%	5%
JavaScript	t_{index}=SO	t_{query}=pp1	86%	12.7%	13.8%	13.9%

Summary of RQ$_3$. In the Experiment II, the lexical function **pp1** was better in all topics ("Java Android", "Java non-Android" and "JavaScript"), i.e., smaller amount of not found results than other preprocessing functions. The column "Not Found" of Table 3 showed that, in some cases, the functions **pp2** and **pp3** had a number of not found results greater than the **raw** function (i.e., for the topics "Java non-Android" and "JavaScript"). The **lsab-java** parser was much better on the topic "Java non-Android" than in the topic "Java Android". One possible explanation for this is that the topic "Java Android" not only uses the Java language for creating graphical user interfaces (GUIs), but also makes use of XML markup language. However, the parser **lsab-java** was not designed for processing XML.

Table 3 shows the results obtained in Experiment II. The column "Not Found" of Table 3 shows the amount of posts from SO not found in each index.

5. EXPERIMENT WITH POTENTIAL API-USAGE-RELATED BUGS

Now, we leverage the knowledge gained in the first three research questions (**RQ$_1$**, **RQ$_2$** and **RQ$_3$**) to study the efficacy of a recommendation system for API-usage-related bugs based on SO. To perform experiments with potential API-usage-related bugs present in different real-world software projects, it was necessary to build a dataset of potential API-usage-related bugs. The construction process of this dataset is a hard task. On SO, users ask different kind of questions. Nasehi et al. [17] identified four categories of questions related to the main concerns of the questioners and what they wanted to solve in SO: *Debug-corrective*, *How-to-do-it*, *Need-to-know* and *Seeking-different-solution*. The *Debug-corrective* category is very close to our work because the developer posts the buggy code in SO in order to glean the fix. But there is a number of bugs that are not related to API methods usage. The next subsection details the steps taken to build this dataset.

5.1 API-usage-related Bugs Dataset

The following steps were performed for the construction of such dataset:

- **Step 1**: We use the 3 filters (contains code in the question, tagged, answered) mentioned above in Section 2 to select candidate API-usage-related posts;
- **Step 2**: A manual analysis was performed with the first 50 returned posts for each programming language, resulting in the identification of 30 API-usage-related bugs, 15 for the Java language and 15 for the JavaScript language;
- **Step 3**: In order to obtain code from real-world software projects that contain potential API-usage-related bugs, we manually analyzed method calls found on snippets from SO posts, selecting the method name related to the respective API-usage-related bug (e.g., a counter-intuitive API method). The corresponding method name was queried in the *OHLOH Code Search* site [12]. We selected the first returned Java class and JavaScript function for being the representative code excerpt with potential API-usage-related bug. In order to produce the final excerpts of Java classes, we removed from the encountered classes the methods that did not call the method used in the query and instance variables.
- **Step 4**: The code excerpts with potential API-usage-related bugs found on the *OHLOH* site were selected to compose the dataset. Then, we construct a pair of two Web addresses for each potential API-usage-related bug present in the dataset: the *OHLOH* address of the code with potential API-usage-related bug and the address of an arbitrary post of SO that fixes this potential API-usage-related bug.

The dataset of potential API-usage-related bugs used in this study is available online [13].

5.2 Experiment III: Excerpts from OHLOH with Potential API-usage-related Bugs

The impact of using real-world code snippets in Experiment III is to simulate a real software maintenance scenario. For each potential API-usage-related bug of the dataset (represented by an excerpt of real open-source code that can lead to API-usage-related bug and its respective correction in SO), we want to measure the position of the Q&A solution in the top-10 results after querying with the real code excerpt. So, for each code searched in our system, the position of the post of SO that solves the potential bug was calculated. It would be a possibility to evaluate the top-10 results for each queried snippet to obtain the quality of the overall recommendation because there may be more than one post of SO that discusses the fix of the potential API-usage-related bug. However, we decided to investigate the position of the API-usage-related bug fix in the ranking to obtain the accuracy of the recommendation.

[12] https://code.ohloh.net/ (verified 05/02/2016)

[13] https://github.com/eduardocunha11/API-usage-related-bugs (verified 30/08/2016)

Table 3: Using SO Snippets as Code-based Queries (SOLR refers to Apache Solr, hence t_{index}=SOLR means that the search index was constructed using Apache Solr, "raw" means that no preprocessing was used). "PP" refers to preprocessing.

Topic	Index PP	Query PP	Not Found	Rank #1	Rank ≤ 5	Rank ≤ 10
Java Android	t_{index}=SOLR$_{raw}$,	t_{query}=raw	15.1%	61.9%	74%	78.5%
Java Android	t_{index}=SOLR$_{pp1}$,	t_{query}=pp1	3.1%	85.7%	94.2%	95%
Java Android	t_{index}=SOLR$_{pp2}$,	t_{query}=pp2	11.1%	71.3%	79%	81.7%
Java Android	t_{index}=SOLR$_{lsab-java}$,	t_{query}=lsab-java	26%	59.9%	68.2%	69.5%
Java non-Android	t_{index}=SOLR$_{raw}$,	t_{query}=raw	11.2%	70.1%	82.4%	85.4%
Java non-Android	t_{index}=SOLR$_{pp1}$,	t_{query}=pp1	3.5%	90%	95%	95.9%
Java non-Android	t_{index}=SOLR$_{pp2}$,	t_{query}=pp2	17%	63.8%	73.9%	76.5%
Java non-Android	t_{index}=SOLR$_{lsab-java}$,	t_{query}=lsab-java	9.9%	79.8%	86.9%	88.7%
JavaScript	t_{index}=SOLR$_{raw}$,	t_{query}=raw	7.3%	74.3%	88.7%	90.4%
JavaScript	t_{index}=SOLR$_{pp1}$,	t_{query}=pp1	0.9%	90.5%	98.1%	98.8%
JavaScript	t_{index}=SOLR$_{pp3}$,	t_{query}=pp3	11.9%	77.5%	85.6%	86.8%
JavaScript	t_{index}=SOLR$_{lsab-js}$,	t_{query}=lsab-js	23.9%	51.6%	58.3%	61.8%

Before explaining how we performed the queries with potential API-usage-related bugs in the indexes of Apache Solr, it is necessary to understand about the keyword filter that was used for query construction. The next subsection presents the methodology used to construct the keyword filter.

5.2.1 Building the Keyword Filter

Table 4 shows the keyword filter used in the Apache Solr's query. The filter is added to the query of Apache Solr at the time which is performed the query in the index. These keywords were taken from a qualitative analysis involving 70 posts of SO containing API-usage-related bugs. These posts were also selected using the 3 filters (contains code in the question, tagged, answered) mentioned above in Section 2. A manual analysis was performed with the first 100 returned posts for each programming language.

Table 4 presents the most recurrent keywords in the titles of these selected posts (i.e., each keyword appeared at least 10 times in the title of these posts). The next subsection explains how to use the keyword filter to query a desired Apache Solr's index.

Table 4: Filter used in the Apache Solr's query.

Keywords considered in the filter
unexpected, incorrect, wrong, output, return, returns, result, results, behavior, weird, strange, odd, problem, problems

5.2.2 Building Queries for Apache Solr using the Keyword Filter

Figure 1 shows the usage of the keyword filter in Apache Solr's query. The example in this figure is for a Java code excerpt #15 . Each document added to the index of Apache Solr has a set of fields, which store the information about document registration. Two fields are shown: *"postTitle"* and *"codeText"*. The former refers to the title of SO's post, while the latter contains the output generated by the preprocessing function after processing the SO code snippet. In this figure, the snippet was preprocessed with the function *lsab-java*, which returns those terms denoting the names of invoked API methods.

The query related to Figure 1 has the following meaning: return the posts of SO (i.e., documents from the index) that have at least one of the keywords defined in Table 4 present in their title and whose code snippets have the API method calls present in the queried code excerpt. The query with no keyword filter would not have the fields *"postTitle"*, but only the field *"codeText"*.

```
((postTitle:unexpected) OR (postTitle:incorrect OR
(postTitle:wrong) OR (postTitle:output) OR
(postTitle:return) OR (postTitle:returns) OR
(postTitle:result) OR (postTitle:results) OR
(postTitle:behavior) OR (postTitle:weird) OR
(postTitle:strange) OR (postTitle:odd) OR
(postTitle:problem) OR (postTitle:problems)) AND
codeText:(LoggedDataInputStream.readLine
readLine dateFormatter.parse parse Date.getTime
getTime Date.getYear getYear Date.setYear
setYear Date.getYear getYear Date.setYear setYear
Date.setYear setYear)
```

Figure 1: Example of Apache Solr's query using the keyword filtering.

For each of the 15 potential API-usage-related bugs present in our dataset selected for the Java language, we performed 6 different queries using Apache Solr, with 2 queries by index (i.e., *pp1* **without filter**, *pp1* **with filter**, *pp2* **without filter**, *pp2* **with filter**, *lsab-java* **without filter** and *lsab-java* **with filter**).

For each of the 15 potential API-usage-related bugs present in our dataset selected for the JavaScript language, we performed 6 different queries using Apache Solr, with 2 queries by index (i.e., *pp1* **without filter**, *pp1* **with filter**, *pp3* **without filter**, *pp3* **with filter**, *lsab-js* **without filter** and *lsab-js* **with filter**). We can see that the first query of each index did not use the keyword filter, while the second query of each index used such filter.

We used the same preprocessing functions used in the Experiment II to conduct the Experiment III. Both the preprocessing functions *pp1* and *raw* suffer from the problem of the presence of reserved words of the language (e.g., stop words as: "public", "class", etc.). Therefore, in Experiment III, the indexes **Java-raw** and **JavaScript-raw** were not considered.

We build new Apache Solr indexes to conduct the Experiment III because we are no longer interested in considering all Java and JavaScript code snippets present in our local

database. Ideally, we would like to index only code snippets related to API-usage-related bugs to improve the search mechanism. To accomplish this, we use the 3 filters (contains code in the question, tagged, answered) mentioned above in Section 2 to select candidate API-usage-related posts. After this, all code snippets present in the post of SO (i.e., snippets present in the question along with snippets present in the answer(s)), after the completion of the preprocessing were indexed so that each post of SO was associated with exactly one document in the index of Apache Solr.

5.2.3 Comparing results with Google

Last but not least, we compared our results to searching in SO with Google, which is the most popular general purpose search engine, i.e., a general front-door to the crowd knowledge.

Other custom code search engines are not suitable for off-the-shelf comparison because they do not index SO content. They would require an adaptation to index snippets and link them to respective posts. In other words, they address a different problem scope.

For each of the 30 potential API-usage-related bugs in our dataset selected for the Java and JavaScript, we performed a Google search within SO aiming at understanding to what extent this search engine retrieves some post of SO mapped in our dataset.

We define a protocol for searching Google, based on the following rules:

- The search query should be restricted to SO site. For this, we specified Google to search within the SO site adding the search parameter "site:stackoverflow.com" to the query;
- Google is not good with long query. Thus, the search query should be limited to a maximum of 32 words (threshold adopted by Google itself).

We also investigate whether Google benefits from querying with preprocessed excerpts. The preprocessing functions *pp1*, *lsab-java*, and *lsab-js* were applied in Java and JavaScript excerpts present in our dataset. We also searched Google without preprocessing those excerpts, i.e., *raw* query.

5.3 Results and Discussion of Experiment III

This section presents and discusses the results obtained in the Experiment III. Table 5 has the following structure: the first column contains the *id* for each OHLOH code excerpt present in our dataset, the second column contains the Web address of the SO's post that fixes a particular potential API-usage-related bug written in Java language. The remaining columns refer to the positions occupied by these posts of SO in the top-10 ranking produced by the Apache Solr search engine and Google search engine. Concerning Apache Solr search engine, we considered the preprocessing functions *pp1*, *pp2*, and *lsab-java* without/with the use of the filter in the query of Apache Solr. Concerning Google, we considered the preprocessing functions *raw*, *pp1*, and *lsab-java*. Each row of the Table 5 corresponds to a potential API-usage-related bug written in Java mapped in our dataset.

Table 6 has similar structure as Table 5 but it is related to potential API-usage-related bugs written in JavaScript language. Concerning Apache Solr search engine, we considered the preprocessing functions *pp1*, *pp3*, and *lsab-js* without/with the use of the filter in the query of Apache Solr. Concerning Google, we considered the preprocessing functions *raw*, *pp1*, and *lsab-js*.

The colored cells of Tables 5 and 6 indicate the positions in the top-10 search results found by Google or Apache Solr. We observed that Google does not cope well in searching for fixes to potential API-usage-related bugs, even with excerpts preprocessed by our parsers (i.e., *lsab-java* and *lsab-js*). These results indicate that these parsers combined with the keyword filter outperform Google in searching for fixes to potential API-usage-related bugs. Those tables show that the likelihood of finding a post of SO that fixes a potential API-usage-related bug written in Java or JavaScript programming languages increases when using the keyword filter in the Apache Solr's query (less "not found" – - cells). Experiment III allowed us to answer the research question $\mathbf{RQ_4}$.

Summary of $\mathbf{RQ_4}$. Experiment III revealed that the proposed approach performed better for Java than for JavaScript. For both languages, the proposed approach obtained the best (and promising) results using, respectively, the preprocessing functions **lsab-java** and **lsab-js**, combined with the keyword filter. Although in general, the proposed approach outperformed Google, the latter could find fixes in specific cases where the approach could not find (four out of thirty cases).

5.4 Threats to Validity

The main threat to external validity is that the dataset we use is small and may not be representative of real world datasets. However, all code excerpts we use are potential API-usage-related bugs present in real software projects. Note that many existing studies evaluate their approaches considering real world code snippets. Moreover, these studies have datasets whose sizes are similar to or much smaller than ours [4][19][11].

Another threat to external validity is that our approach is as good as the database of SO. Currently, results are presented based on the data dump of March 2013. Although, the database of SO is growing rapidly. There are approximately 12M questions in the database today (June 2016), while in March 2013, the database had about 3M questions.

The main threat to internal validity is the size of code snippets with potential API-usage-related bugs queried in our system. Our intuition is that the larger the size of the code snippet queried in our system, the greater is the probability of introducing noise in the query results because a larger number of API methods could be considered (i.e., different API methods could confuse the proposed approach). On the other hand, our approach is a flexible solution because it allows developers to inform their own suspicious snippets of different sizes and get extra knowledge from SO discussions.

6. RELATED WORK

API-usage-related bugs are not really studied in the literature.

Crowd Debugging. Gao et al. [4] proposed a fully automatic approach to fixing recurring crash bugs via analyzing Q&A sites. Their approach extracts queries from crash traces and retrieves a list of Q&A pages. They investigated recurring bugs (i.e., bugs that occur often in different projects) and observed that many recurring bugs have already been

Table 5: Results of queries performed in the Java indexes *pp1*, *pp2*, and *lsab-java* (without/with the use of the keyword filter in the Apache Solr's query; "-" = Post of SO not found in the ranking; "P" refer to the positions occupied by posts of SO in the ranking.

Id	Post of SO that fixes the potential API-usage-related bug	*Solr Without Filter*			*Solr With Filter*			*Google*		
	http://stackoverflow.com/questions/	P_{pp1}	P_{pp2}	P_{lsab}	P_{pp1}	P_{pp2}	P_{lsab}	P_{raw}	P_{pp1}	P_{lsab}
1	13896614/surprising-result-from-math-pow65-17-3233	-	-	-	-	-	-	-	-	-
2	9065727/why-long-tohexstring0xffffffff-returns-ffffffffff	-	-	2	-	-	1	-	-	3
3	12975924/math-cos-java-gives-wrong-result	-	-	1	-	-	1	1	-	2
4	10603336/bigdecimal-floor-rounding-goes-wrong	-	-	-	-	-	9	-	-	-
5	11597244/why-the-result-of-integer-tobinarystring-for	-	-	-	6	-	5	-	-	2
6	6899019/java-simpledateformat-returns-unexpected	-	-	-	6	2	-	-	-	-
7	9956471/wrong-result-by-java-math-pow	-	-	-	-	-	-	4	-	-
8	12175674/is-java-math-bigintegers-usage-wrong	2	-	2	2	1	1	-	-	-
9	1755199/calendar-returns-wrong-month	-	-	-	8	-	-	-	-	-
10	12213877/is-identityhashmap-class-wrong	-	-	1	-	-	1	-	-	-
11	14102593/unexpected-output-with-iso-time-8601	-	-	-	-	-	6	-	-	-
12	9230961/unexpected-output-converting-a-string-to-date	-	-	-	-	-	-	-	-	-
13	14213778/unexpected-result-with-outputstream-in-java	-	-	-	-	8	6	-	-	-
14	14836004/java-date-giving-the-wrong-date	-	-	-	-	9	3	-	-	-
15	7215621/why-does-javas-date-getyear-return-111	-	-	-	-	4	3	-	-	-

Table 6: Results of queries performed in the JavaScript indexes *pp1*, *pp3* and *lsab-js* (without/with the use of the keyword filter in the Apache Solr's query; "-" = Post of SO not found in the ranking; "P" refer to the positions occupied by posts of SO in the ranking.

Id	Post of SO that fixes the potential API-usage-related bug	*Solr Without Filter*			*Solr With Filter*			*Google*		
	http://stackoverflow.com/questions/	P_{pp1}	P_{pp3}	P_{lsab}	P_{pp1}	P_{pp3}	P_{lsab}	P_{raw}	P_{pp1}	P_{lsab}
16	12318830/parseint08-returns-0	-	-	-	-	-	-	3	-	-
17	9859995/unexpected-output-in-javascript	-	-	-	-	-	5	-	-	-
18	11477415/why-does-javascripts-regex-exec-not-always	-	-	-	-	-	-	-	-	-
19	8979093/json-stringifyobject-incorrect	-	-	-	-	-	-	-	-	-
20	2587345/javascript-date-parse	-	-	-	-	-	-	-	-	-
21	834757/why-does-getday-return-incorrect-values	-	-	5	3	9	1	-	2	-
22	1845001/array-sort-is-not-giving-the-desired-result	-	-	-	-	-	-	-	-	-
23	9766201/unexpected-behavior-during-subtraction	-	-	-	-	-	1	-	-	-
24	8160550/unexpected-result-adding-numbers-79-0014-95	-	-	-	-	-	-	-	-	-
25	11363526/weird-output-of-97-98-mapstring-fromcharcode	-	-	-	2	-	-	2	-	-
26	6223616/math-cos-and-math-sin-are-inaccurate	-	-	2	-	-	-	-	-	-
27	8524933/json-parse-unexpected-character-error	-	-	-	-	-	2	-	-	-
28	10706272/unexpected-javascript-date-behavior	-	-	-	7	1	2	-	-	-
29	18509996/get-index-of-empty-space-returns-wrong-value	-	-	-	-	-	9	-	-	-
30	262427/javascript-arraymap-and-parseint	-	-	-	-	-	-	3	1	1

discussed over the Q&A sites such as SO. Our approach is designed to handle a particular class of recurring bugs called API-usage-related bugs. The latter are related to incorrect usage of an API by a developer and do not generate exception stack traces. Moreover, we proposed to search SO using snippet-based queries instead of terms extracted from exception stack traces.

Monperrus et al. [15] defined the concept of "crowd bugs" (i.e., synonym for "API-usage-related bugs") and proposed the idea of "debug with the crowd". Our approach was based on Monperrus et al. paper and present a complete new set of empirical results. Moreover, we investigated API-usage-related bugs present in Java and JavaScript code snippets, while Monperrus et al. only handle JavaScript API-usage-related bugs.

F. Chen and S. Kim [2] proposed to mine SO in order to help developers during debugging tasks. Their approach is aimed at detecting different types of bugs. Unlike them, our approach is designed to provide fixes and explanations from SO for a particular type of bug (i.e., API-usage-related bugs). Our approach differs in the sense that we suppose a scenario where developers already know the suspicious snippet and wants to find if there is a discussion on that pattern of snippet.

Hartmann et al. [6] presented *HelpMeOut*, a social recommender system that aids the debugging of compiler error messages by suggesting solutions that other programmers have applied in the past. *HelpMeOut* uses both error messages and source code context in the capture and search for relevant fixes. Unlike *HelpMeOut*, our approach uses only code snippets to search SO for API-usage-related bug fixes because API-usage-related bugs do not generate exception stack traces.

Semantic code search and code search engines. The main usage of code search engines is to retrieve code samples and reusable open source code from the Web. Different works tackled this problem [32][20][25][26]. The mining of open source repositories has also been used to identify API and framework usage and to find highly relevant applications to be reused [13][14][27]. Differently from the work done so far on code search, we do not target open source repositories to provide code samples and reusable code, or to understand the usage of APIs; instead, we target the crowd knowledge provided by discussions in SO as alternative source. This is because we want to provide developers with API-usage-related bug fixes with explanations, rather than just with reusable code components/snippets.

Recommender Systems. Different typologies of recommender systems to suggest relevant project artifacts, support newcomers in a project, and assist developers by suggesting code examples on the Web relevant to their current task has been presented. Well-known examples are HIPIKAT

[30], DEEPINTELLISENCE [7], and eROSE [34]. Other work focused on suggesting relevant documents, discussions and code samples from the Web to fill the gap between the IDE and the Web browser. Examples are CODETRAIL [5], MICA [23], FISHTAIL [22], and DORA [9].

Among the various sources available on the Web, Q&A Websites and in particular SO, have been the target of many recommender systems. In a previous work [1], we present an approach that makes use of "crowd knowledge" available in SO to recommend information that can assist developers in their development tasks with a given API. This strategy recommends a ranked list of question-answer pairs from SO based on a query. The criteria for ranking are based on three main aspects: the textual similarity of the pairs with respect to the query related to the developer's problem, the quality of the pairs, and a filtering mechanism that uses machine learning to consider only "how-to" posts. However, there are some differences between the work presented here and this previous work: (i) the previous approach is more general and it was not designed to deal with API-usage-related bugs; (ii) the previous approach relies on searching Lucene indexes using a list of terms as input query, while the current approach uses suspicious code snippets as input query; (iii) the previous approach was designed to assist developers during development tasks while the current approach was designed to assist developers during debugging tasks.

Ponzanelli et al. [19] proposed *Prompter*, a recommender system designed to automatically capturing the code context in the IDE and suggesting documents from SO that have enough self-confidence to the developer. Our approach also takes into account the developer's code context with the aim of suggesting relevant SO discussions. However, there are some differences between the two works: (i) their approach is more general and addresses a different problem scope (i.e., they do not limit their scope to "API-usage-related bugs"); (ii) they do not use the Apache Solr as search engine. Instead, they use other search engines (i.e., Google, Bing, and Blekko); (iii) they implement a different code searching technique.

7. CONCLUSIONS

In this paper, we presented a new approach to find fixes for API-usage-related bugs using suspicious code snippets as input query. Typical general and source code search engines are not designed to handle this type of query. For all experiments, we considered preprocessing functions for code snippets written in Java or JavaScript languages. We built a dataset composed by 30 excerpts with potential API-usage-related bugs extracted from *OHLOH* in order to evaluate our approach. The results showed a clear advantage in approximately 40% of the approach *using the filter* in relation to the approach *without using the filter*. Moreover, the results are encouraging: for 66.67% of Java excerpts with potential API-usage-related bugs picked from OHLOH, we found their fix in the top-10. Concerning JavaScript excerpts present in the dataset, for 40% of them, we found their fix in the top-10. These results suggest that our approaches (i.e., *lsab-java* and *lsab-js* combined with the keyword filter) outperform Google and SO in searching for fixes to potential API-usage-related bugs present in real software projects.

8. ACKNOWLEDGMENTS

We would like to thank the Brazilian agencies FAPEMIG, CAPES and CNPq for partially funding this research.

9. REFERENCES

[1] E. C. Campos, L. B. L. de Souza, and M. d. A. Maia. Searching Crowd Knowledge to Recommend Solutions for API Usage Tasks. *Journal of Software: Evolution and Process*, 2016. Wiley.

[2] F. Chen and S. Kim. Crowd debugging. In *Proceedings of the ESEC/FSE' 2015*. ACM.

[3] L. B. L. d. Souza, E. C. Campos, and M. d. A. Maia. On the Extraction of Cookbooks for APIs from the Crowd Knowledge. In *Brazilian Symposium on Software Engineering (SBES' 2014)*, pages 21–30, Sept 2014.

[4] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In *Proc. 30th ASE*, pages 307–318, 2015.

[5] M. Goldman and R. C. Miller. Codetrail: Connecting Source Code and Web Resources. *Journal of Visual Languages and Computing*, 20(4):223–235, Aug. 2009.

[6] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the CHI '10*, pages 1019–1028. ACM, 2010.

[7] R. Holmes and A. Begel. Deep Intellisense: A Tool for Rehydrating Evaporated Information. In *Proc. of the MSR '08*, pages 23–26, New York, USA, 2008. ACM.

[8] S. Kim and E. J. Whitehead, Jr. How Long Did It Take to Fix Bugs? In *Proc. of the MSR '06*, pages 173–174, New York, NY, USA, 2006. ACM.

[9] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes. Automatically locating relevant programming help online. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC' 2012)*, pages 127–134, Sept 2012.

[10] C. W. Krueger. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, Jan. 2012.

[12] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using Stack Overflow. In *Proceedings of the MSR' 2013*, pages 93–96. IEEE.

[13] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Trans. Software Eng.*, 38(5):1069–1087, 2012.

[14] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proceedings of the 33rd ICSE '11*, pages 111–120, New York, NY, USA, 2011. ACM.

[15] M. Monperrus and A. Maia. Debugging with the Crowd: a Debug Recommendation System based on Stackoverflow. Technical report, INRIA, 2014.

[16] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How Can I Use This Method? In *Proc. of the ICSE '15*, pages 880–890. IEEE Press.

[17] S. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example? A study of programming Q&A in Stack Overflow. In *Proceedings of the ICSM' 2012*, pages 25–34.

[18] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. ISSTA '2011*, pages 199–209.

[19] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the MSR' 2014*, pages 102–111, NY, USA. ACM.

[20] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[21] M. P. Robillard and R. Deline. A Field Study of API Learning Obstacles. *Empirical Softw. Engg.*, 16(6):703–732, Dec. 2011.

[22] N. Sawadsky and G. C. Murphy. Fishtail: From task context to source code examples. In *Proceedings of the 1st Workshop on Developing Tools As Plug-ins*, pages 48–51, New York, NY, USA, 2011. ACM.

[23] J. Stylos and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proc. of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202. IEEE Computer Society, 2006.

[24] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API Documentation. In *Proceedings of the ICSE '2014*, pages 643–652, New York, NY, USA. ACM.

[25] S. Thummalapenta. Exploiting Code Search Engines to Improve Programmer Productivity. OOPSLA '07, pages 921–922. ACM, 2007.

[26] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. 22nd ASE*, pages 204–213, 2007.

[27] S. Thummalapenta and T. Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *Proc. 23rd ASE*, pages 327–336, 2008.

[28] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? In *Proc. of the ICSE '11*, pages 804–807. ACM.

[29] C. Treude and M. P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *Proc. of the ICSE '2016*, pages 392–403. ACM.

[30] D. Čubranić and G. C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. of the 25th ICSE '03*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[31] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the MSR '2006*, pages 54–57, New York, NY, USA. ACM.

[32] A. Zagalsky, O. Barzilay, and A. Yehudai. Example Overflow: Using social media for code recommendation. In *Proc. of the RSSE' 2012*, pages 38–42. IEEE Computer Society.

[33] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring Resource Specifications from Natural Language API Documentation. In *Proc. 24th ASE*, pages 307–318, 2009.

[34] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th ICSE '04*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.