A Comment-Driven Approach to API Usage Patterns Discovery and Search

Shin-Jie Lee[1,2], Xavier Lin[2] and Wu-Chen Su[1]
[1] Computer and Network Center, National Cheng Kung University, Taiwan
[2] Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
Email: jielee@ mail.ncku.edu.tw, XavierLinX@gmail.com, wuchen_sue@hotmail.com

***Abstract*—API usage patterns have been considered as significant materials in reusing software library APIs for saving development time and improving software quality. Although efforts have been made on discovering API usage patterns, how to enable a programmer to search the discovered usage patterns is still largely unexplored. This paper presents a comment-driven approach to discovering and searching API usage patterns with two key features: first, API usage patterns are discovered with keywords through mining the comments in open source projects; second, the discovered usage patterns can be searched by natural language queries based on semantic similarities. In the experimental evaluations, 1775 API usage patterns are discovered from 10510 open source projects. The precision of the patterns search is 83% that is significantly higher than that of a traditional keyword match-based code search system.**

Keywords—API usage patterns discovery and search, code examples

I. INTRODUCTION

Reusing existing software code has been considered as a significant activity in software development for saving development time and improving software quality [1][2][3][4][5][6][7][8]. To date, there are several information sources from which a programmer can retrieve API usage code examples, including peers [9], API documentations [29][30], web sites with pre-collected code examples [12][13], and code search engines [14][15][16].

It has been found that many programming learners rely on their more experienced peers for finding code examples to overcome API selection barriers [9]. Other than acquiring the code from peers, a programmer could find code examples from API documentations [29][30]. Although code examples are regarded as an important factor in designing API documentations [10][11], it has been argued that most API documents lack code examples because manually crafting code examples for all API methods is labor intensive and time consuming [19]. Apart from API documentations, there are web sites [12][13] that provide a set of pre-collected code examples. However, managing and maintaining a set of code examples rely heavily upon a tremendous amount of collection effort or the contents contributed by the web site users. In addition, there are code search engines that enable programmers to search the code snippets in the open source code in terms of file names, classes, methods or structures based on traditional keyword matching [14][15][16].

Although efforts have been made on discovering API usage patterns or examples in a more systematic way [18][19][20][21][23][24][28], little emphasis has been put on how to enable a programmer to search the discovered API usage patterns. The key challenge can be best explained as: *How to search API usage patterns if a programmer does not know the API method names?* In most approaches, it is assumed that programmers exactly know what API methods [18][19][21][23][27] or the input/output types [17][26] they are going to use and then use the API method names as queries in searching relevant usage patterns. The assumption would limit the possibilities of finding out more usage patterns that meet the programmers' needs.

In this work, we propose a comment-driven approach to discovering and searching API usage patterns with the following two key features: First, each discovered API usage pattern is associated with multiple keywords and tf-idf values through mining the comments in open sources. In the proposed approach, code snippets and the associated comments in open sources are extracted through a proposed algorithm. Based on the extracted code snippets and comments, API usage patterns are discovered together with associated keywords. Second, a programmer is supported to search the discovered API usage patterns by free form natural language queries based on the semantic similarities between the queries and the keywords, and the similarities between the queries and related comments.

The remainder of the paper is organized as follows: Section II contains a review of related work. Section III fully describes the proposed approach. Experimental evaluations are discussed in Section IV. Finally, in Section V, we summarize the contributions of the proposed approach.

II. RELATED WORK

Several approaches have been proposed for discovering or searching relevant code snippets or API usage patterns.

Ohloh Code Search [16] is an on-line code search engine for more than 20 billion lines of open source code. For a natural language query with multiple terms, the search engine returns a number of code snippets of open source code that contains the query terms.

Mandilin et al. [17] proposed an approach to synthesizing code snippets for a query that is described by the desired code in terms of input and output types. The synthesized code snippets are ranked by their lengths. The assumption of the approach is

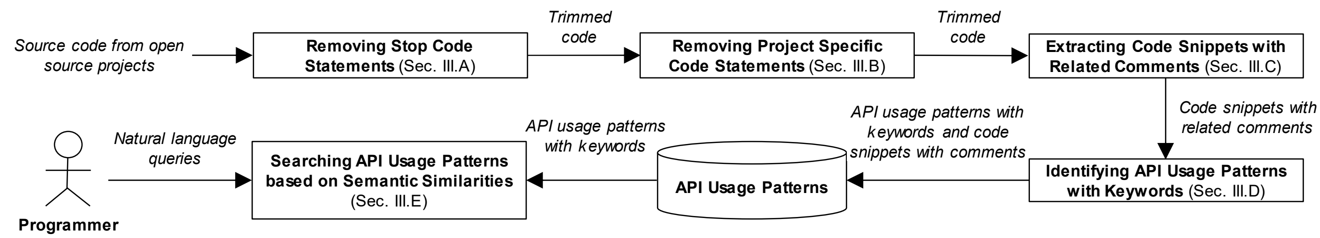

Fig. 1. Operational concept of the comment-driven approach to API usage patterns discovery and search.

that a programmer knows what type of object he needs but does not know how to write the code to get the object.

Holmes et al. [18] proposed an approach to locating relevant code in a code example repository based on heuristically matching the structure of the code under development as a query to the code examples. The code examples that occur most frequently in the set generated from applying all of the heuristics are selected and returned to the programmer for the query.

Kim et al. [19] proposed a code example recommendation system that provides API documents embedded with high-quality code example summaries mined from the Web. In the system, code examples from an existing code search engine are summarized into code snippets. Subsequently, the semantic features of the summarized code snippets are extracted and the most representative code examples will be automatically identified while a user chooses an API from the API documents.

Zhong et al. [21] developed a framework, called MAPO, for mining API usage patterns from open source repositories. API usage patterns are discovered based on a frequent subsequence miner [22], and are ranked based on the similarities between class and method names containing the supporting snippets and the ones containing the specified method to be used by the programmer. The programmer can further exploit the source code of each code snippet of an API usage pattern.

Wang et al. [23] proposed an approach, called UP-Miner, to mining succinct and high-coverage usage patterns of API methods from source code. Given a user-specified API method, UP-Miner can automatically search for all usage patterns of an API method and return associated code snippets as candidates for reusing.

Chatterjee et al. [20] proposed a code search technique, call SNIFF, that retains the flexibility of performing code search in plain English. The key idea of SNIFF is to combine API documentations with publicly available Java code. It takes a large amount of Java source code and annotates it by appending each statement containing a method call with the description of the method in Java API documentations (Javadoc). The annotated source code are then indexed for free-form query search.

TABLE 1 summarizes the technical features of the related work. In most approaches, it is assumed that programmers exactly know what API methods they want to use and then use the API methods as queries in searching relevant code snippets or API usage patterns. The assumption is relaxed in SNIFF approach by accommodating the searches with free-form natural language queries based on annotations derived from Javadoc. However, its approach is hard to locate code snippets if the terms of a query are not contained in the related Javadoc. By contrast, our approach analyzes a large number of comments written by variant developers from open source code rather than Javadoc, and explore more possible keywords that are often used by the programmers but are not contained in Javadoc. Furthermore, a programmer is supported to use natural language queries or API method names to search API usage patterns based on the comments in open source code written by various developers.

III. A COMMENT-DRIVEN APPROACH TO API USAGE PATTERNS DISCOVERY AND SEARCH

Fig. 1 shows the operational concept of the comment-driven approach to API usage patterns discovery and search. Firstly, source code from a set of collected open source projects is parsed, and the following two types of code statements will be removed: (1) stop code statement: a code statement that is extremely common and is hardly considered as parts of an API usage pattern; and (2) project specific code statement: a code statement that consists of invocations of methods written in the same project of the code statement. In this work, a stop code statement or a project specific code statement is a line of code. Secondly, a number of code snippets and the related comments will be extracted from the trimmed source code. Thirdly, based on the code snippets and the comments, API usage patterns and their associated keywords will be identified. The API usage patterns, keywords, code snippets and comments are then stored in a repository. Finally, a programmer can search API usage patterns by natural language queries based on a proposed semantic similarity formula.

TABLE 1. COMPARISON OF THE RELATED WORK

	Ohloh Code Search [16]	Mandelin et al. [17]	Holmes et al. [18]	Kim et al. [19]	Zhong et al. [21]	Wang et al. [23]	SNIFF [20]	The Proposed Approach
Queries	Natural language queries	Input and output types	Code under development	Specified API methods	Specified API methods	Specified API methods	Natural language queries	Natural language queries or specified API methods
Search Results	Code snippets containing the query terms	Synthesized code snippets with the input and output types	Code snippets that heuristically match the query	Representative code snippets	API usage patterns with associated code snippets	API usage patterns with associated code snippets	Relevant code snippets	API usage patterns with exemplary code snippets

TABLE 2. STOP CODE STATEMENTS

#	Code Statement	Frequency
1	System.out.println	65,367
2	System.out.printf	54,801
3	System.err.println	16,389
4	System.out.print	6,173
5	System.exit	5,100

TABLE 3. AN EXAMPLE OF A CODE SNIPPET CONTAINING STOP CODE STATEMENTS

1:	Score score = new Score();
2:	//serialize the Score object
3:	ByteArrayOutputStream out = new ByteArrayOutputStream();
4:	ObjectOutputStream oout = new ObjectOutputStream(out);
~~5:~~	~~System.out.println("Serializing…");~~
6:	oout.writeObject(score);
7:	oout.flush();
8:	oout.close();
9:	score.reset();

TABLE 4. AN EXAMPLE OF A CODE SNIPPET CONTAINING PROJECT SPECIFIC CODE STATEMENTS

~~1:~~	~~Score score = new Score();~~
2:	//serialize the Score object
3:	ByteArrayOutputStream out = new ByteArrayOutputStream();
4:	ObjectOutputStream oout = new ObjectOutputStream(out);
~~5:~~	~~System.out.println("Serializing…");~~
6:	oout.writeObject(score);
7:	oout.flush();
8:	oout.close();
~~9:~~	~~score.reset();~~

A. Removing Stop Code Statements

From the open source projects, we have observed that there are some code statements that are extremely common and are hardly considered as parts of an API usage pattern. For example, the statement System.out.println(arguments) is often used by programmers for logging or debugging and does not contribute to system functionalities in most cases. In this paper, these statements are called *stop code statements*.

Through parsing the source code of 1,000 Java open source projects from sourceforge.net, frequencies of the code statement that satisfies both of the following two criteria will be obtained: (1) The code statement describes an invocation of a method in ignore of its arguments; and (2) The object to be invoked in criteria 1 is not a local variable. The first criterion constrains that the code statement describes an invocation of an API method, and the second criterion ensures that the code statement does not contain a local variable that may be with different identifier names in different projects.

TABLE 2 shows the top 5 frequently appeared code statements. In this work, these code statements are selected as stop code statements. In order to increase the effectiveness of discovering API usage patterns, the stop code statements will be removed from the open source code. TABLE 3 shows an example of a code snippet that contains a stop code statement in line 5. After removing the line of the stop code statement, line 6 will be the next line of line 4.

B. Removing Project Specific Code Statements

As an API usage pattern is supposed to recurrently appear across projects, it should not contain invocations of the methods that are particularly written for one project. In this work, a code statement is called a project specific code statement if it consists

TABLE 5. ALGORITHM OF EXTRACTING CODE SNIPPETS WITH RELATED COMMENTS

1:	**let** P be a set containing all projects
2:	**let** F be a set containing all source code files of P after removing stop and project specific code statements
3:	**let** $S = \varnothing$ // to collect code snippets
4:	**let** $R = \varnothing$ // to collect regularized code snippets
5:	**let** $C = \varnothing$ // to collect comments
6:	**let** $rgl : S \rightarrow R$, $cmt : S \rightarrow C$, $file: S \rightarrow F$, $prj: S \rightarrow P$
7:	**for each** source code file $f \in F$ of project $p \in P$ **do**
8:	Extract all comments in the bodies of the methods in f
9:	**for each** comment c **do**
10:	**let** s be a code snippet directly following c
11:	**let** s_i be the first i lines of s ($1 \leq i \leq n$, and n is the number of lines of s)
12:	**let** s_i^r be a regularized form of s_i
13:	**for** $i = n$ to 1 **do**
14:	**let** $S = S \cup \{s_i\}$
15:	**let** $C = C \cup \{c\}$
16:	**let** $rgl(s_i) = s_i^r$, $cmt(s_i)=c$, $file(s_i)=f$, $prj(s_i)=p$
17:	**if** $s_i^r \in R$ **then**
18:	**break**
19:	**else**
20:	**let** $R = R \cup s_i^r$
21:	**end if**
22:	**end for**
23:	**end for**
24:	**end for**

of invocations of methods that are written in the same project of the code statement, and will also be removed from the source code to improve the effectiveness of discovering API usage patterns.

For example, it is assumed that Score is a class written in the project of the code snippet of TABLE 3. As line 1 is to invoke the constructor method of Score and line 9 is to invoke the method reset of the score object, they are considered as project specific code statements and will be removed from the source code (see TABLE 4).

If a method invocation is used as an argument of another method invocation, it will be bypassed while parsing the open source code. For example, if line 4 is changed to be ObjectOutputStream oout = new ObjectOutputStream (score.getOutputStream()), it will not be considered as a project specific code statement although it contains an invocation of the method getOutputStream of the score object. The invocation will be regularized as a uniform name while extracting code snippets and comments (see Section III.C).

C. Extracting Code Snippets with Related Comments

After the stop and project specific code statements are removed from the open source code, a large number of code snippets and their related comments will be extracted from the trimmed source code by a proposed algorithm with the following steps (see TABLE 5).

First (lines 1-5), the trimmed source code files F of all projects P are prepared, and empty sets S, R and C are declared to collect code snippets, regularized code snippets and comments, respectively.

Second (line 6), four functions *regl*, *cmt*, *file* and *prj* are defined to relate a code snippet to a regularized code snippet, a comment, a file and a project, respectively.

Third (lines 7-8), all comments in the bodies of the methods in every source file $f \in F$ of project $p \in P$ are extracted by parsing the source code. In this work, Eclipse JDT API is used for parsing the Java source files.

Forth (lines 9-10), for each comment c, the code snippet s that directly follows c is extracted. A code snippet directly follows a comment if it conforms to the pattern expressed in TABLE 6. The code snippet is right after the comment without any lines in the middle of them and is followed by a blank line or another comment.

Fifth (lines 11-12), multiple code snippets $s_1, \ldots, s_n$ are created based on the code snippet s of n lines of code, where s_i is the first i lines of s. Each code snippet s_i is then *regularized* as a new code snippet s_i^r through the following two activities: (1) replacing all arguments of method invocations, variables, identifiers and literals by a uniform name _VAR, and (2) adjusting the code to a uniform format. TABLE 7 shows the regularized code snippets $s_1^r, \ldots, s_5^r$ that directly follow the comment in line 2 of TABLE 4.

At last (lines 13-22), each code snippet s_i is iteratively obtained in the order of $s_n, \ldots, s_1$ and added into set S. Meanwhile, the comment c is added into the comments set S. s_i is related to the regularized code snippet s_i^r, the comment c, the file f and the project p by the four functions. After that, if s_i^r has already been included in the regularized code snippets set R, the iteration will be broken out, otherwise s_i^r will be added into R. This step is to explore more code snippets that are related to the comment and increase the chances of finding out more API usage patterns.

D. Identifying API Usage Patterns with Keywords

Based on the extracted code snippets and the related comments, API usage patterns with keywords will be discovered. An API usage pattern is defined as follows:

Definition 1 (API Usage Pattern). *An API usage pattern r is a regularized code snippet that recurrently appears in multiple projects, and API usage patterns are defined as a set*

$$AP = \{r \mid r \in R;\ and\ |P_r| \geq k\},$$

where $P_r = \{p \mid p \in P;\ p = prj(s);\ and\ s \in S_r\}$ *is the set of projects in which the pattern r appears, and k is the minimum number of projects in which an API pattern appears.* $S_r = \{s \mid s \in S;\ and\ rgl(s) = r\}$ *denotes the set of code snippets of regularization form r.*

In order to better discover API usage patterns that are frequently used by various programmers, a regularized code snippet is identified as an API usage pattern if it recurrently appears in multiple (more than k) projects. Because a code snippet may be copied and pasted multiple times in the source code of the same project by a programmer, the number of appearances of an API usage pattern in a project is not considered in the identifications of patterns.

Once an API usage pattern r is identified, several words will be identified as the keywords of the pattern from the comments by the following steps:

1. Generate a document d_r related to an API usage pattern r by aggregating the comments to which the code snippets of regularized form r relate. Meanwhile, any two comments that are with edit distance [25] ≤ 3 will be removed in order to filter out the comments originally created by copy-paste with little changes. The document is formally defined as $d_r = \{c \mid c \in C;\ c = cmt(s);\ and\ s \in S_r\}$, and the documents for all the patterns are defined as $D = \{d_r \mid r \in AP\}$.
2. Remove stop words and stem the words in document d_r. Stop words are extremely common words and are usually omitted in natural language processing (NLP) systems [31]. Some stop words are *the, is, are, at*, and *below*. Stemming a word is to transform the word into its part of the word that is common to all its inflected variants. For instance, *creates* and *created* are stemmed as *creat*.
3. Calculate the tf-idf value of each word in d_r. tf-idf (term frequency-inverse document frequency) formula [31] is widely used to reflect the importance of a word in a document. In this work, the formula serves as a basis for identifying the keywords of an API usage pattern. The *term frequency* of a term t^r in d_r is calculated as the raw frequency of t^r in d_r divided by the sum of the raw frequencies of all terms in d_r:

$$tf_{t^r} = \frac{f(t^r, d_r)}{\sum_{w \in D} f(w, d_r)} \tag{1}$$

The *inverse document frequency* of t^r is to measure how the term is common or rare across all documents in D, and is calculated by dividing the total number of documents in D by the number of documents containing t^r plus 1, and then taking the logarithm of the quotient:

TABLE 6. PATTERN OF A CODE SNIPPET THAT DIRECTLY FOLLOWS A COMMENT

1:	Code, or a blank line	
2:	**comment**	comment *c*
3:	**comment**	
4:	Code	code snippet *s* directly following *c*
5:	Code	
6:	Code	
7:	A blank line, or a comment	

TABLE 7. AN EXAMPLE OF EXTRACTED REGULARIZED CODE SNIPPETS AND THEIR RELATED COMMENT

Comment		
	// serialize the Score object	
Code snippet directly following the comment		**Extracted Code Snippets**
1:	ByteArrayOutputStream out = new ByteArrayOutputStream();	s_1
2:	ObjectOutputStream oout = new ObjectOutputStream(out);	s_2
3:	oout.writeObject(score);	s_3
4:	oout.flush();	s_4
5	oout.close();	s_5
Regularized code snippet		**Extracted Regularized Code Snippets**
1:	ByteArrayOutputStream _VAR = new ByteArrayOutputStream();	s_1^r
2:	ObjectOutputStream _VAR = new ObjectOutputStream(_VAR);	s_2^r
3:	_VAR.writeObject(_VAR);	s_3^r
4:	_VAR.flush();	s_4^r
5	_VAR.close();	s_5^r

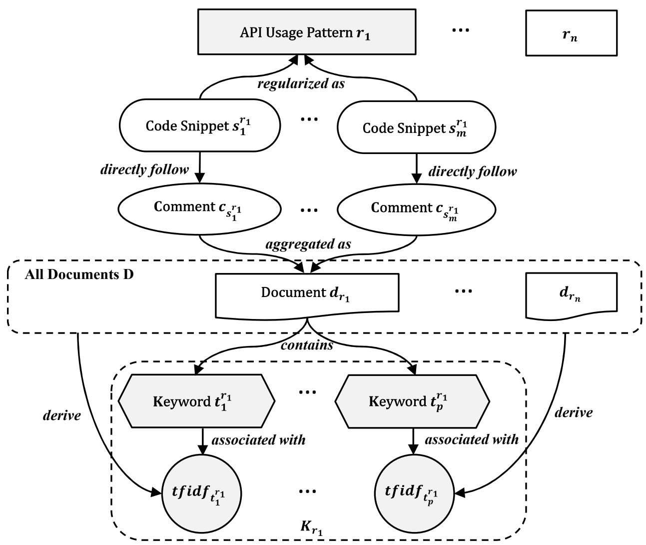

Fig. 2. Relationships between API usage patterns, keywords, code snippets, and comment.

$$idf_{t^r} = \log\left(\frac{|D|}{1+|d \in D : t^r \in d\,|}\right) \qquad (2)$$

The tf-idf value of t^r is calculated by the following equation

$$tfidf_{t^r} = tf_{t^r} \times idf_{t^r} \qquad (3)$$

4. Identify the terms of top p tf-idf values in d_r as the keywords of the API usage pattern r:

$$K_r = \{t | t \in d_r;\ t\ is\ with\ a\ top\ p\ tf\text{-}idf\ value\ \}.$$

Fig. 2 shows the relationships between API usage patterns, keywords, code snippets, and comments. $r_1, \dots, r_n$ are API usage patterns. r_1 is the regularization form of code snippets $s_1^{r_1}, \dots, s_m^{r_1} \in S_{r_1}$ that directly follow comments $c_{s_1^{r_1}}, \dots, c_{s_m^{r_1}}$ respectively. The comments are aggregated as a document d_{r_1} that contains keywords $t_1^{r_1}, \dots, t_p^{r_1}$ with tf-idf values derived from documents $d_{r_1}, \dots, d_{r_n}$.

E. Searching API Usage Patterns based on Semantic Similarities

After API usage patterns are discovered, a programmer will be enabled to search API usage patterns by natural language queries based on vector space model (VSM) [32].

Definition 2 (Semantic Similarity between a Natural Language Query and an API Usage Pattern). *Let A and B be two sets of terms with tf-idf values. The cosine similarity between A and B is calculated as*

$$sim(A, B) = \frac{\sum_{t \in A \cup B} tfidf_{t,A} \times tfidf_{t,B}}{\sqrt{\sum_{t \in A} tfidf_{t,A}{}^2} \times \sqrt{\sum_{t \in B} tfidf_{t,B}{}^2}}. \qquad (4)$$

Let q be a natural language query consists of a set of terms and r be an API usage pattern. The semantic similarity between q and r is calculated as

$$similarity_{q,r} = sim(q, K_r) \times max\{sim(q, c_{s_i}r)\} \quad (5),$$

TABLE 8. AN EXAMPLE OF THE SEMANTIC SIMILARITY BETWEEN AN API USAGE PATTERN AND A NATURAL LANGUAGE QUERY

Query q	**Keywords of usage pattern r**	**Comment $c_{s_1}r$**	**Comment $c_{s_2}r$**	**Comment $c_{s_3}r$**
serialize/0.6	serialize/0.3	unmarshal/0.5	serialize/0.6	serialize/0.6
byte/0.5	byte/0.1	object/0.3	object/0.3	message/0.4
array/0.4	marshal/0.1	file/0.2	array/0.4	array/0.5
$similarity_{q,r} = 0.79 \times \max\{0, 0.75, 0.73\} = 0.59$				

where the term frequencies (tf values) of terms in query q and comment $c_{s_i}r$ are calculated as the raw frequencies of the terms in q and $c_{s_i}r$ divided by the sums of the raw frequencies of all terms in q and $c_{s_i}r$, respectively. The inverse document frequencies (idf values) of the terms are calculated by Equation 2.

In equation 5, $sim(q, K_r)$ will be calculated as a higher value if the important terms in *q* are the same as the ones in the keywords of *r*. The second operand $\max\{sim(q, c_{s_i}r)\}$ will be calculated as a higher value if there is a comment associated with *r* whose important terms are the same as the ones in *q*. The semantic similarity between q and r ranges from 0 meaning independence to 1 meaning exactly the same.

TABLE 8 shows an example of the semantic similarity between an API usage pattern and a natural language query. The query is "serialize to byte array", where "to" is a stop word and hence is removed. The tf-idf values of "serialize", "byte" and "array" are calculated as 0.6, 0.5 and 0.4, respectively. The keywords of an API usage pattern *r* are "serialize", "byte" and "marshal" with tf-idf values 0.3, 0.1 and 0.1, respectively. In addition, there are three code snippets of the usage pattern with different comments $c_{s_1}r$, $c_{s_2}r$ and $c_{s_3}r$. The semantic similarity between the usage pattern and the query is then calculated as $0.79 \times \max\{0, 0.75, 0.73\} = 0.59$, and s_2 will be selected as the exemplary code snippet of the usage pattern through Definition 3.

Definition 3 (Exemplary Code Snippet with a Query). *Let an API usage pattern r be the regularization form of code snippets $s_1^r, \dots, s_m^r$ that directly follow comments $c_{s_1^r}, \dots, c_{s_m^r}$ respectively. The exemplary code snippet of r for the query q is the code snippet that directly follows the comment of the following semantic similarity value with q:*

$$max\{sim(q, c_{s_1}r), \dots, sim(q, c_{s_m}r)\}.$$

The programmer will be provided with an exemplary code snippet together with a comment that is most semantically similar to the query. The selected comment serves as an informative content for the programmer in determining whether to adopt the searched API usage pattern with the query. Therefore, different code snippets may be selected as examples for the same API usage pattern with different queries.

The search result for a query is a list of ranked API usage patterns according to the semantic similarities between the query and the patterns. In addition, a threshold can be given to filter out the usage patterns with low similarity values for the query. A searched API usage pattern will be presented together with an exemplary code snippet.

TABLE 9. PAIRED SAMPLES T-TEST FOR COMPARING PRECISIONS OF THE PROPOSED APPROACH AND OHLOH CODE SEARCH SYSTEM

#	Query	Ohloh Code Search			The Proposed Approach		
		$\lvert Ret\rvert_{\#}$	$\lvert Rel\rvert_{\#}$	$Precision_{10}$	$\lvert Ret\rvert_{\#}$	$\lvert Rel\rvert_{\#}$	$Precision_{10}$
1	add a sleep thread	10	2	**0.2**	3	3	**1.00**
2	Add a White Background	10	8	**0.8**	3	3	**1.00**
3	Add action listeners to menu items	10	4	**0.4**	10	8	**0.80**
4	add all files in dir	10	6	**0.6**	2	0	**0.00**
5	Add buttons to the panel	10	9	**0.9**	10	10	**1.00**
6	add components on main panel using GridBagLayout	10	0	**0**	1	1	**1.00**
7	add task to thread pool	10	4	**0.4**	1	1	**1.00**
8	Add the error Message Label	10	4	**0.4**	5	5	**1.00**
9	add this key to the map	10	2	**0.2**	1	1	**1.00**
10	build a 255 white spaces string	10	10	**1**	2	0	**0.00**
11	build a source map from the arguments as key-value pairs	10	4	**0.4**	1	1	**1.00**
12	build a transparent image	10	9	**0.9**	2	2	**1.00**
13	build vector	10	3	**0.3**	1	1	**1.00**
14	Building URL	10	5	**0.5**	1	1	**1.00**
15	Button with Icon	10	5	**0.5**	3	3	**1.00**
16	Check for existence of directory	10	7	**0.7**	10	8	**0.80**
17	Check if project exists	10	5	**0.5**	8	3	**0.38**
18	check if the file is a xml file	10	4	**0.4**	3	3	**1.00**
19	Check the file exists	10	7	**0.7**	10	10	**1.00**
20	Checkbox	10	3	**0.3**	5	3	**0.60**
21	close input stream	10	10	**1**	10	3	**0.30**
22	close the connection at once	10	2	**0.2**	3	2	**0.67**
23	connect to the database	10	6	**0.6**	3	2	**0.67**
24	convert set to string array	10	7	**0.7**	5	4	**0.80**
25	Create a coordinate for a point	10	3	**0.3**	2	2	**1.00**
26	create an ObjectOutputStream	10	8	**0.8**	1	1	**1.00**
27	Create char array and copy chars	10	3	**0.3**	2	2	**1.00**
28	Create directory for class file if it does not exist.	10	6	**0.6**	10	9	**0.90**
29	create hash	10	6	**0.6**	1	1	**1.00**
30	Create menu	10	3	**0.3**	10	10	**1.00**
31	Create panel	10	3	**0.3**	10	10	**1.00**
32	create project	10	1	**0.1**	9	4	**0.44**
33	Create the 3D canvas	10	6	**0.6**	1	0	**0.00**
34	create the file	10	0	**0**	5	3	**0.60**
35	create the pool and start the threads now	10	1	**0.1**	2	2	**1.00**
36	Create the zip entry and add it to the jar file	10	3	**0.3**	2	2	**1.00**
37	create toolbar	10	6	**0.6**	5	4	**0.80**
38	create XML document	10	3	**0.3**	10	9	**0.90**
39	Delete data files	10	4	**0.4**	4	4	**1.00**
40	Deserialize from byte array.	10	8	**0.8**	2	0	**0.00**
41	draw 1 pixel border	10	9	**0.9**	5	4	**0.80**
42	draw Image	10	9	**0.9**	9	3	**0.33**
43	draw line	10	4	**0.4**	9	5	**0.56**
44	draw Text	10	9	**0.9**	10	6	**0.60**
45	Fill in background	10	5	**0.5**	7	7	**1.00**
46	Fill the bitmap with the current background color	10	4	**0.4**	2	2	**1.00**
47	First close the network connectors to stop accepting new connections	2	1	**0.5**	1	1	**1.00**
48	Flush the buffer	10	0	**0**	3	3	**1.00**
49	free memory	10	6	**0.6**	2	1	**0.50**
50	Get all of the projects in the workspace	10	1	**0.1**	6	5	**0.83**
51	Get source file names.	10	1	**0.1**	1	1	**1.00**
52	Get the icon width and height	10	5	**0.5**	3	2	**0.67**
53	getting name and extension	10	4	**0.4**	4	4	**1.00**
54	hide the dialog	10	8	**0.8**	1	1	**1.00**
55	HTML Tail Section	10	10	**1**	1	1	**1.00**
56	HTTP header contents	10	8	**0.8**	1	1	**1.00**
57	ignore not directory	10	1	**0.1**	10	6	**0.60**
58	image size	10	10	**1**	5	4	**0.80**
59	In reverse order	2	1	**0.5**	2	2	**1.00**
60	Load into a temporary buffer and create another input stream	10	3	**0.3**	1	1	**1.00**
61	load properties	10	9	**0.9**	8	7	**0.88**
62	Load the contents of our configuration file	10	3	**0.3**	3	2	**0.67**
63	now stop the server	10	8	**0.8**	2	1	**0.50**
64	Now write the data back to the file	10	9	**0.9**	6	6	**1.00**
65	ok and cancel buttons	10	2	**0.2**	10	6	**0.60**
66	only want to read first 10 bytes...	4	0	**0**	1	1	**1.00**
67	Open dialog	10	6	**0.6**	4	4	**1.00**
68	open file	10	1	**0.1**	7	3	**0.43**
69	Open the editor for this file.	10	3	**0.3**	4	2	**0.50**
70	paint horizontal lines	10	3	**0.3**	1	1	**1.00**
71	parse root element	10	9	**0.9**	3	3	**1.00**
72	parse the command line arguments	10	5	**0.5**	2	2	**1.00**
73	place the zip entry in the ZipOutputStream object	10	8	**0.8**	1	1	**1.00**
74	prepare File	10	1	**0.1**	2	1	**0.50**
75	Prepare icons.	10	8	**0.8**	1	1	**1.00**
76	Prepare workspace	10	0	**0**	1	1	**1.00**
77	read a string	10	5	**0.5**	2	2	**1.00**
78	read data	10	6	**0.6**	7	7	**1.00**
79	Read file and write outputs	10	4	**0.4**	4	4	**1.00**
80	read next byte from input stream	10	8	**0.8**	2	2	**1.00**
81	Read next n lines	10	9	**0.9**	2	2	**1.00**
82	Read the file	10	2	**0.2**	10	10	**1.00**
83	select root element	10	2	**0.2**	4	4	**1.00**
84	serialize the Object	10	4	**0.4**	10	10	**1.00**
85	Serialize to byte array.	10	3	**0.3**	7	7	**1.00**
86	Set tab width	10	3	**0.3**	2	2	**1.00**
87	Set the database connection	10	8	**0.8**	3	2	**0.67**
88	show description as a tooltip	10	5	**0.5**	1	1	**1.00**
89	Show dialog for selecting a directory	10	10	**1**	3	1	**0.33**
90	sort the array	10	8	**0.8**	2	1	**0.50**
91	sort the list	10	4	**0.4**	2	2	**1.00**
92	space check	10	0	**0**	1	1	**1.00**
93	start the worker thread and leave it running	10	8	**0.8**	3	3	**1.00**
94	Status label	10	10	**1**	6	6	**1.00**
95	strip quotes	10	8	**0.8**	3	3	**1.00**
96	temporary file to use	10	3	**0.3**	4	4	**1.00**
97	Test existing file	10	4	**0.4**	9	8	**0.89**
98	Test file is a directory	10	7	**0.7**	10	6	**0.60**
99	transfer input stream to output stream, via a buffer	10	7	**0.7**	2	2	**1.00**
100	Write out the string	10	8	**0.8**	3	2	**0.67**
	Average Precision:			**0.51**			**0.83**
	t=7.579 (>1.984) (With 95% confidence level and degrees of freedom=99)						

If the query is the name of an API method, usage patterns containing invocations of the method will be selected and ranked by the number of projects they appear.

IV. EXPERIMENTAL EVALUATION

We implemented an API usage patterns search tool as an Eclipse plugin. Fig 3 shows a snapshot of searching API usage patterns. With this tool, a programmer is enabled to search API usage patterns by typing a comment and pressing “Ctrl+6”. A list of searched API usage patterns will be presented with highlighted exemplary code snippets. The programmer can browse the code snippets and then copy-paste the snippets into their code.

In order to better reflect the real search queries that may be used by programmers, we extract all the comments from another 10 projects that are not used for discovering usage patterns, and then use the extracted comments as queries to search API usage patterns. In the experiment, the threshold for

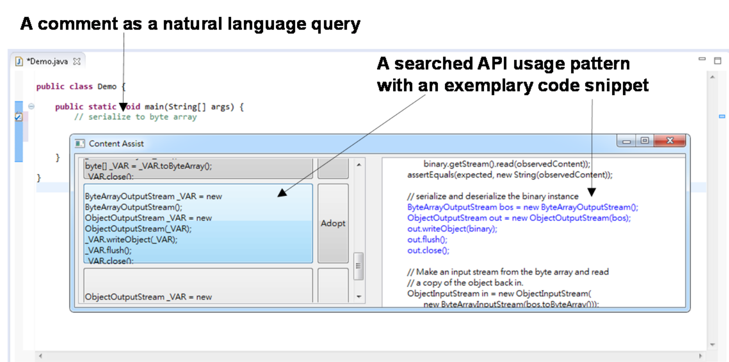

Fig. 3. A snapshot of searching API usage patterns

the semantic similarities between the query and the usage patterns is set to 0.2. We filtered out the queries with no returns, and then randomly selected 100 queries from the rest (see

TABLE 10. EXAMPLES OF THE API USAGE PATTERNS SEARCH RESULTS

Query q	API usage pattern r	Top 20 keywords	Exemplary code snippet	Comment of the exemplary code snippet	$similarity_{q,r}$ (≥0.2)

Transfer input stream to output stream, via a buffer	byte[] _VAR = new byte[_VAR];
	int _VAR;
	while ((_VAR = _VAR.read(_VAR)) > _VAR) {
	  _VAR.write(_VAR, _VAR , _VAR);
	}
	_VAR.close();
		transfer, byte, instream, outstream, bo, gzip, wise, kb, copi, bi, chunk, out, bit, destin, file, stream, store, sourc, output, input	byte[] buf = new byte[1024];
			int len;
			while ((len = in.read(buf)) > 0) {
			  out.write(buf, 0, len);
			}
			in.close();
				// Transfer bytes from the input file to the gzip output stream	0.214

Getting name and extension	String _VAR = _VAR ;
	String _VAR = _VAR.getName();
	int _VAR = _VAR.lastIndexOf(_VAR);
	if (_VAR > _VAR && _VAR < _VAR.length() - _VAR) {
	  _VAR = _VAR.substring(_VAR + _VAR).toLowerCase();
	}
		extens, recuper, find, dot, file, part, last, de, first, name	String ext = null;
			String s = f.getName();
			int i = s.lastIndexOf('.');
			if (i > 0 && i < s.length() - 1) {
			  ext = s.substring(i + 1).toLowerCase();
			}
				//Get Extension	0.36

Load the contents of our configuration file	Properties _VAR = new Properties();
	FileInputStream _VAR = new FileInputStream(_VAR);
	_VAR.load(_VAR);
	_VAR.close();
		properti, load, launcher, lit, prop, fichier, dan, file, param, le, pass, variabl, la, configur, try, name	Properties properties = new Properties();
			FileInputStream inputStream = new FileInputStream(configurationFile);
			properties.load(inputStream);
			inputStream.close();
				// Load the properties from configuration file	0.249

Strip quotes	if (_VAR.startsWith(_VAR) && _VAR.endsWith(_VAR)) {
	  _VAR = _VAR.substring(_VAR , _VAR.length() - _VAR);
	}
		quot, end, start, string, cut, remov, strip, apici, doppi, scicoslabg, findvalu, esterni, vera, mine, vedo, causa, quotat, ascii, truncat, di	if (boundary.startsWith("\"") && boundary.endsWith("\"")) {
			  boundary = boundary.substring(1, boundary.length() - 1);
			}
				// Strip off quotes if provided	0.387

Serialize to byte array	ByteArrayOutputStream _VAR = new ByteArrayOutputStream();
	ObjectOutputStream _VAR = new ObjectOutputStream(_VAR);
	_VAR.writeObject(_VAR);
	_VAR.flush();
	_VAR.close();
		serial, recreat, deseri, byte, arrai, marshal, binari, object, pass, instanc, sourc, except, write, default, out, data, valu	ByteArrayOutputStream out = new ByteArrayOutputStream();
			ObjectOutputStream oout = new ObjectOutputStream(out);
			oout.writeObject(epr1);
			oout.flush();
			oout.close();
				// serialize	0.425

Add a sleep thread	try {
	  Thread.sleep(_VAR);
	} catch (InterruptedException _VAR) {
	  _VAR.printStackTrace();
	}
		wait, sleep, give, time, thread, ms, littl, bit, second, work, until, delai, sec, paus, finish, bridg, pdu, cpu, befor, tomcat	try {
			  Thread.sleep(100);
			} catch (InterruptedException e) {
			  e.printStackTrace();
			}
				// sleep this thread until there is something to read	0.305

TABLE 9). The search results of the 100 selected queries are manually reviewed by three Java experts. The precision of the search result of each query is calculated by the following definition:

Definition 4 (Precision of API Usage Patterns Search). *Given a natural language query q, the precision of the search result is calculated as*

$$Precision_n = \frac{|Rel|_\#}{|Ret|_\#}, \quad (8)$$

where Ret denotes top n returned API usage patterns, and Rel denotes the usage patterns (∈Ret) that are relevant to q.

An API usage pattern is determined to be relevant to a query if all the three experts consider it is relevant to the query. With $n = 10$, the precisions of the proposed approach for the 100 queries are calculated and the average precision is 83%. The average precision of Ohloh Code Search is 51%. Fig 4 shows the Boxplot of the precisions of Ohloh and the proposed approach. The t-value for comparing the precisions is 7.579 (>1.984) with degrees-of-freedom=99 and 95% confidence level, which indicates that there is a significant difference between the average values of the precisions of the two approaches. As there may be more than 10 relevant results in the API usage patterns database and the Ohloh database, the recalls are not measured in the experiment.

TABLE 10 shows the examples of the API usage patterns search results. For the first query, the term "transfer" does not appear in the Java documentations of the API methods read(), write() and close(), but is identified as the top 1 keyword for the usage pattern by the proposed approach. Therefore, the usage pattern will be considered to be semantically similar to the query with degree 0.214 (≥ threshold 0.2), and then will be returned.

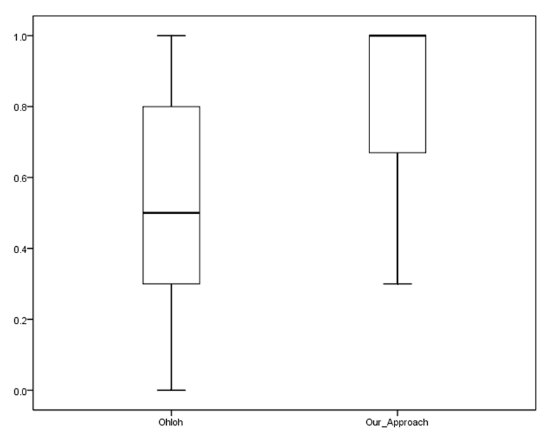

Fig. 4. Boxplot showing the precisions of Ohloh and the proposed approach

Similarly, although the terms "extension", "configuration" and "strip" do not appear in the Java documentations of the API methods in the usage patterns for the second, third and fourth queries, respectively, they are identified as keywords of the usage patterns for searches. For the fifth query, the terms "serialize", "byte" and "array" are identified as top 5 keywords of the API usage pattern that consists of a sequence of method invocations: new ByteArrayOutputStream(), new ObjectOutputStream(), writeObject(), flush(), and close(). For the last query, an API usage pattern including try-catch statements that are commonly used for Thread.sleep will be returned.

V. Conclusion

The contribution of this paper is twofold: (i) API usage patterns are discovered together with keywords mined from the comments in a large number of open sources through a proposed algorithm; and (ii) programmer is supported to search the discovered API usage patterns by free form natural language queries based on the semantic similarities between the queries and the keywords, and the ones between the queries and related comments. In addition, we conducted experiments to validate the proposed approach. It has been validated that the precision of the API usage patterns search is significantly higher than the one of a traditional keyword match-based code search system. In the future, we will improve the current implementation with the following directions: (i) mining more open source projects to collect more terms related to code snippets; (ii) extending the synonym database to include more synonyms; and (iii) providing spell checking feature.

Acknowledgment

This research is sponsored by Ministry of Science and Technology under the grant 103-2221-E-006-218 in Taiwan.

References

[1] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, Jul. 2005.

[2] T. Ravichandran and M. A. Rothenberger, "Software Reuse Strategies and Component Markets," *Communications of the ACM*, vol. 46, no. 8, pp. 109-114, Aug. 2003.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM*, vol. 39, no. 10, pp. 104-116, Oct. 1996.

[4] S. Haefliger, G. von Krogh, and S. Spaeth, "Code Reuse in Open Source Software," *Management Science*, 54(1), pp. 180-193, 2008.

[5] M.L. Griss, "Software Reuse: Architecture, Process and Organization for Business Success," *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pp. 86-89, 1997.

[6] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The State of the Art in End-User Software Engineering," *ACM Computing Surveys*, vol. 43, no. 3, Article 21, 2011.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[8] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling Reusable Software out of Thin Air," *IEEE Software*, vol. 25, issue 5, pp. 45-52, 2008.

[9] A. J. Ko, B. A. Myers, and H. H. Aung, "Six Learning Barriers in End-User Programming Systems," *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 199-206, 2004.

[10] W. Maalej and M. P. Robillard, "Patterns of Knowledge in API Reference Documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264-1282, 2013.

[11] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, vol. 26, no. 6, pp. 27-34, 2009..

[12] http://www.java2s.com/

[13] Microsoft Developer Network. http://code.msdn.microsoft.com/

[14] Krugle Search. http://opensearch.krugle.org/

[15] Merobase Component Finder. http://www.merobase.com/

[16] Black Duck Software, "Ohloh Code Search," http://code.ohloh.net/.

[17] D. Mandelin, L. Xu, R. BodíK, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle, "*Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 48-61, 2005.

[18] R. Holmes, R. J. Walker and G. C. Murphy, "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952-970, Dec. 2006.

[19] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching Documents with Examples: A Corpus Mining Approach," *ACM Transactions on Information Systems*, vol. 31, no. 1 , pp. 1-27, 2013.

[20] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A Search Engine for Java Using Free-Form Queries," *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pp. 385-400, 2009.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns, "*Proceedings of the 23rd European Conference on ECOOP - Object-Oriented Programming*, pp. 318-343, 2009.

[22] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential Pattern Mining Using a Bitmap Representation," *Proceedings of 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 429-435, 2002.

[23] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining Succinct and High-Coverage API Usage Patterns from Source Code," *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pp. 319-328, 2013.

[24] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, issue 4, pp. 80-86, 2010.

[25] D. Jurafsky and J. H. Martin, "Speech and Language Processing," *Pearson Education International*, pp. 107–111, 1966.

[26] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pp. 204-213, 2007.

[27] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for Sample Code," *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pp. 413-430, 2006.

[28] T. T. Nguyen, H. A. Nguyen, and N. H. Pham, "Graph-based Mining of Multiple Object Usage Patterns," *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 383-392, 2009.

[29] Java API. http://docs.oracle.com/javase/7/docs/api/.

[30] MSDN Library. http://msdn.microsoft.com/en-us/library.

[31] A. Rajaraman and J. D. Ullman, "Mining of Massive Datasets," *Cambridge University Press*, Dec. 30, 2011.

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