JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 593-610 (2018)
DOI: 10.6688/JISE.201805_34(3).0002

Methods for Categorizing and Recommending API Usage Patterns Based on Degree Centralities and Pattern Distances*

SHIN-JIE LEE[1,3], WU-CHEN SU[2], CHI-EN HUANG[3] AND JIE-LIN YOU[3]
[1]*Computer and Network Center*
[3]*Department of Computer Science and Information Engineering*
National Cheng Kung University
Tainan, 701 Taiwan
[2]*Department of Clinical Sciences*
University of Kentucky
Lexington, KY 40506 USA
E-mail: jielee@mail.ncku.edu.tw; wuchen_sue@hotmail.com; {tony4794; ygl0118}@gmail.com

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 and searching API usage patterns, the following two issues are still largely unexplored: how to provide a well-organized view of the discovered API usage patterns? and how to recommend follow-up API usage patterns once a usage pattern is adopted? This paper proposes two methods for categorizing and recommending API usage patterns: first, categories of the usage patterns are automatically identified based on a proposed degree centrality-based clustering algorithm; and second, follow-up usage patterns of an adopted pattern are recommended based on a proposed metric of measuring distances between patterns. In the experimental evaluations, the patterns categorization can achieve 85.4% precision rate with 83% recall rate. The patterns recommendation had approximately half a chance of correctly predicting the follow-up patterns that were actually used by the programmers.

Keywords: API usage pattern, code example, code search system, API usage patterns categorization, API usage patterns recommendation

1. INTRODUCTION

In recent years, a number of approaches have been proposed for discovering or searching relevant code snippets or API usage patterns. Open Hub [1] 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 that contain the query terms. Mandilin *et al.* [2] 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. Holmes *et al.* [3] 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.

Kim *et al.* [4] proposed a code example recommendation system that provides API

Received January 28, 2017; revised March 8, 2017; accepted March 9, 2017.
Communicated by Ren-Hung Hwang.
* This research is sponsored by Ministry of Science and Technology under the grant 105-2221-E-006-154-MY3 in Taiwan.

documents embedded with high-quality code example summaries mined from the Web. Chatterjee *et al.* [5] 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. Zhong *et al.* [6] 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 [7], and are ranked based on the similarities between class and method names. Wang *et al.* [8] proposed an approach, called UP-Miner, to mining succinct and high-coverage usage patterns of API methods from source code. Subramanian *et al.* [9] proposed an iterative, deductive method of linking source code examples to API documentation for increasing the timeliness of the API documentation. In [10], Janjic *et al.* discussed the foundations of software search and reuse, and provided the main characteristics of reuse-oriented code recommendation (ROCR) systems. GitHub Gists [11] is an on-line website where users can share their code snippets in single files, parts of files, or full applications. Users can search a number of code examples with natural language queries. Code Recommenders [12] is an Eclipse plugin, and one of its features is Snipmatch which provides a way to search for code snippets.

Although efforts have been made on discovering [3-8, 13-15] and searching [2-4, 6, 8, 15-19] API usage patterns or examples, little emphasis has been put on how to automatically categorize and recommend follow-up usage patterns, which can be best explained as follows: (1) *How to provide a well-organized view of the discovered API usage patterns?* In most code examples web sites [20, 21], code examples are grouped into categories to provide programmers with a well-organized view for browsing and selecting API usage code examples. However, how to automatically categorize the discovered API usage patterns through a more automatic way is still a challenge; (2) *How to recommend follow-up API usage patterns once a usage pattern is adopted?* It is a tedious work for a programmer to come up with next relevant queries once an API usage pattern is selected from the search results or a category, which poses a challenge of predicting follow-up usage patterns according to the adopted usage pattern.

In this work, we propose two methods based on our previous work [22] to address these challenges:

1. **A proposed degree centrality-based clustering algorithm** A programmer is provided with a set of API usage pattern categories that are automatically identified based on a proposed degree centrality-based clustering algorithm. Based on the keywords of the API usage patterns, the usage patterns are clustered into categories together with automatically identified tags to better provide the programmer with a well-organized view of the usage patterns for selections and adoptions.
2. **A proposed metric of measuring distances between API usage patterns** A number of recommended follow-up API usage patterns for each adopted usage pattern are provided based on a proposed metric of measuring distances between API usage patterns. The distance between two API usage patterns is calculated based on the distances between their associated code snippets. Once an API usage pattern is adopted from the search results or a category, a number of usage patterns close to the usage pattern will be recommended for writing follow-up code.

In the experiments, we present the statistics of discovered API usage patterns and a developed API usage pattern assistant tool. Additionally, we also evaluate the precision and recall of categorizing API usage patterns, and the hit rate of recommending following-up API usage patterns. The remainder of the paper is organized as follows: Section 2 fully describes the proposed methods. Section 3 discusses the experimental evaluations. Finally, in Section 4, we summarize the contributions of the proposed methods.

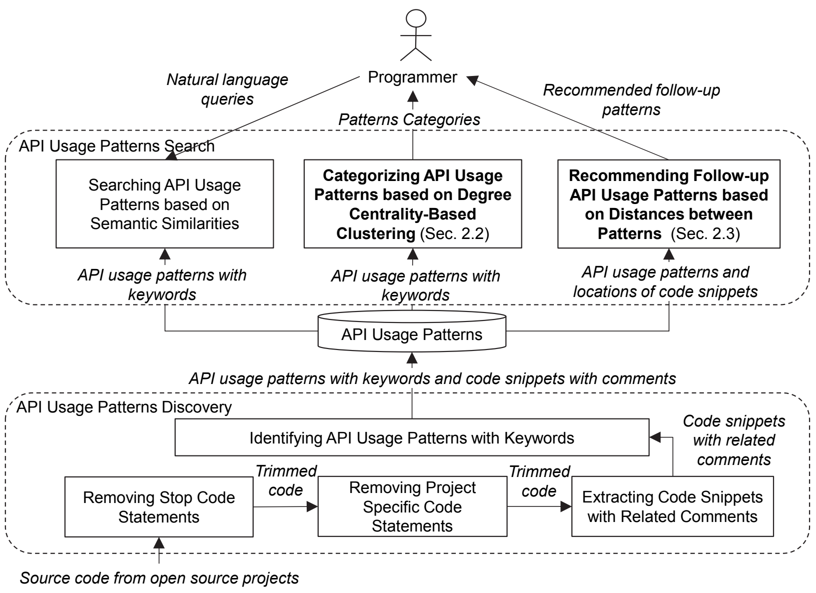

Programmer
Natural language queries
Patterns Categories
Recommended follow-up patterns
API Usage Patterns Search
Searching API Usage Patterns based on Semantic Similarities
Categorizing API Usage Patterns based on Degree Centrality-Based Clustering (Sec. 2.2)
Recommending Follow-up API Usage Patterns based on Distances between Patterns (Sec. 2.3)
API usage patterns with keywords
API usage patterns with keywords
API usage patterns and locations of code snippets
API Usage Patterns
API usage patterns with keywords and code snippets with comments
API Usage Patterns Discovery
Identifying API Usage Patterns with Keywords
Code snippets with related comments
Removing Stop Code Statements
Trimmed code
Removing Project Specific Code Statements
Trimmed code
Extracting Code Snippets with Related Comments
Source code from open source projects

Fig. 1. System architecture of the extended API usage pattern discovery and search system.

2. THE PROPOSED METHODS FOR CATEGORIZING AND RECOMMENDING API USAGE PATTERNS

In our previous work [22], we developed an API usage patterns discovery and search system through which a programmer can search API usage patterns by natural language queries based on a proposed semantic similarity formula. The definition of a discovered API usage pattern is described in Section 2.1. In this work, the system was extended with two key features: a) the API usage patterns are clustered into categories based on a proposed degree centrality-based clustering algorithm to provide the programmer with a well-organized view of the patterns (Section 2.2); and b) once an API usage pattern is selected from the search results or a category, the subsystem will recommend the programmer a set of API usage patterns that may be used in writing the follow-up code based on distances between patterns (Section 2.3).

Fig. 1 shows the system architecture of the extended API usage pattern discovery and search system. The system can be broken down into two subsystems: API usage patterns discovery and API usage patterns search.

In the API usage patterns discovery subsystem, 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. After that, a number of code snippets and the related comments will be extracted from the trimmed source code. 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.

In the API usage patterns search subsystem, a programmer can search API usage patterns by natural language queries based on a proposed semantic similarity formula. In addition, the API usage patterns are clustered into categories based on a proposed degree centrality-based clustering algorithm to provide the programmer with a well-organized view of the patterns. Once an API usage pattern is selected from the search results or a category, the subsystem will recommend the programmer a set of API usage patterns that may be used in writing the follow-up code based on distances between patterns.

2.1 Definition of API Usage Patterns

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

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

where $P_r=\{p|p\in P; p=prj(s); \text{ and } s\in S_r\}$ 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. $S_r=\{s|s\in S; \text{ and } rgl(s)=r\}$ 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 [24] ≤ 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 $d_r=\{c|c\in C; c=cmt(s); \text{ and } s\in S_r\}$, and the documents for all the patterns are defined as $D=\{d_r|r\in AP\}$.

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 [25]. 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 [26] 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:

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

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

   $$tfidf_{t^r} = tf_{t^r} \times idf_{t^r}. \tag{3}$$

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

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

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

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). At first, given two sets, namely A and B, of terms with tf-idf values, the cosine similarity between A and B is calculated as $sim(A,B) = \frac{\sum_{t \in A \cup B} tfidf_{t,A} \times tfidf_{t,B}}{\sqrt{\sum_{t \in A} tfidf_{t,A}^2 \times \sum_{t \in B} tfidf_{t,B}^2}}$.

Definition 2 (Semantic Similarity between a Natural Language Query and an API Usage Pattern) 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

$$similarity_{q,r} = sim(q, K_r) \times \max\{sim(q, c_{s_i^r})\}. \tag{4}$$

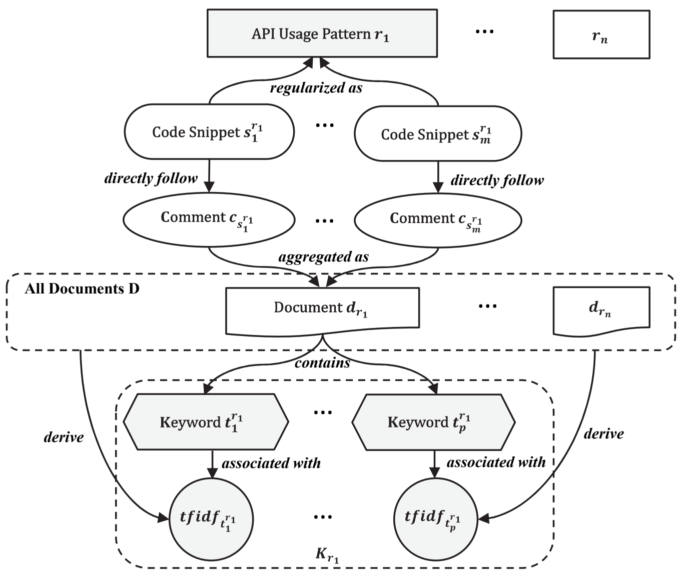


Fig. 2. Relationships between API usage patterns, keywords, code snippets, and comments.

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 by Eq. (2). $sim(q, K_r)$ 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.

2.2 Categorizing API Usage Patterns based on Degree Centrality-Based Clustering

In this work, the API usage patterns are automatically clustered into categories through a proposed clustering algorithm based on the concept of degree centrality [23] to provide a programmer with a well-organized view of the patterns, which is defined as follows:

Definition 3 (Degree Centrality of an API Usage Pattern): Let $G = (AP, E)$ be a weighted and undirected cosine similarity graph comprising a set API usage patterns $\in AP$ as nodes together with a set weighted edges E that connect any two patterns to each other. The weight of an edge $e_{r_i,r_j} \in E$ of the nodes r_i and r_j is calculated as

$$weight_{e_{r_i,r_j}} = \frac{\sum_{t\in K_{r_i}\cup K_{r_j}} tfidf_{t,K_{r_i}} \times tfidf_{t,K_{r_j}}}{\sqrt{\sum_{t\in K_{r_i}} tfidf_{t,K_{r_i}}^{\,2}} \times \sqrt{\sum_{t\in K_{r_j}} tfidf_{t,K_{r_j}}^{\,2}}}. \tag{5}$$

The degree centrality of an API usage pattern r is defined as the degree of r in the similarity graph G_h corresponding to a threshold $h\in[0,1]$, where $G_h = (AP, E_h)$, and $E_h =$

$\{e|e\in E$; and $(1 - weight_e) \leq h\}$. K_{r_i} and K_{r_j} denote the keywords of the API usage pattern r_i and r_j. G_h is a sub-graph of G. The value of $similarity(K_{r_i}, K_{r_j})$ represents the semantic similarity between r_i and r_j. Fig. 3 shows an example of the degree centrality of an API usage pattern. Two patterns are placed closely if they are with a high semantic similarity value. As there are five edges of r_1 with the weights that conform to the constraint $(1 - weight_e) \leq h$, the degree centrality of r_1 is 5.

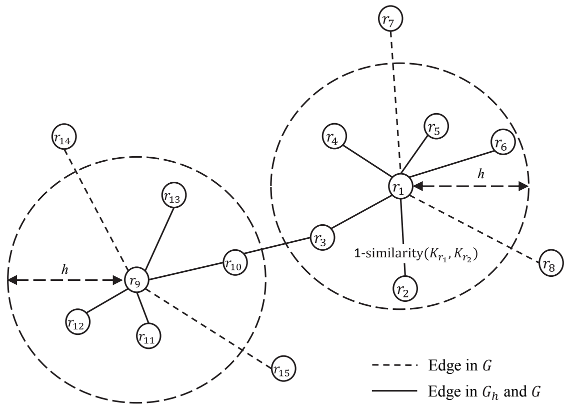


Fig. 3. An example of two clusters of API usage patterns. The degree centrality scores of patterns r_1 and r_9 are 5 and 4, respectively.

Based on the degree centralities of API usage patterns, an algorithm is proposed to cluster the API usage patterns with the following steps (see Table 1): First (lines 1-2), let $G_h = (AP, E_h)$ be a similarity graph corresponding to a threshold h, and $CLSTS$ be an empty set to collect clusters of API usage patterns. Second (lines 3-4), the API usage pattern r with the highest degree centrality is selected if G_h contains one or more nodes. Third (lines 5-6), r together with all its neighbors in G_h are grouped as a cluster cl_r, and r serves as the central node of the cluster. The cluster is then added into $CLSTS$. At last (lines 7-8), all the API usage patterns in cl_r are removed from G_h, and the iteration repeats until all nodes are removed from G_h.

In Fig. 3, for example, pattern r_1 is selected at first as a central node because it is with the highest degree centrality 5 among the nodes in G_h, and then r_1 together with its five neighbors r_2, …, r_6 are grouped into a cluster. After that, r_1, …, r_6 are removed from G_h, and r_9 will be the next central node with the highest degree centrality 4 among the nodes in G_h. r_9, …, r_{13} are then grouped into another cluster. In this work, a cluster is regarded as a category of API usage patterns.

Table 1. Degree centrality-based clustering algorithm for categorizing API usage patterns.

```
1:  let $G_h = (AP, E_h)$ be a similarity graph
2:  let $CLSTS = \emptyset$   //to collect clusters
3:  do while $G_h$ contains one or more nodes
4:      Select the API usage pattern with the highest degree centrality $r$ from $AP$
5:      let $cl_r$ = {All neighbors of $r$ in $G_h$} $\cup$ {$r$}
6:      $CLSTS = CLSTS \cup cl_r$
7:      Remove nodes of $cl_r$ from $G_h$
8:  end do
```

In addition, a category will be automatically tagged with a set of keywords as the identifier of the category.

Definition 4 (Tags of an API Usage Patterns Category): Let *cl* be a cluster containing API usage patterns $r'_1, \dots, r'_n$. The category tags of *cl* are defined as a set

$$Tag_{cl} = \{t \mid t \in d_{cl}; \text{ and } t \text{ is with a top 3 tf-idf value}\}.$$

d_{cl} denotes the document consisting of documents $d_{r'_1}, \dots, d_{r'_n}$. The term frequency of a term $t^{cl} \in d_{cl}$ is calculated as the raw frequency of t^{cl} in d_{cl} divided by the sum of the raw frequencies of all terms in $d_r : tf_{t^{cl}} = \frac{f(t^{cl}, d_{cl})}{\sum_{w \in D} f(w, d_{cl})}$. The inverse document frequency of t^{cl} is calculated by $idf_{t^{cl}} = \log(\frac{|D|}{1+|d \in D: t^{cl} \in d|})$, where $D=\{d_r \mid r \in AP\}$. The tf-idf value of t^{cl} is calculated as $tfidf_{t^{cl}} = tf_{t^{cl}} \times idf_{t^{cl}}$.

In order to better reflect the keywords of a category as tags rather than an API usage pattern, all comments associated with the patterns in the category are considered as a whole, d_{cl}, in calculating the term frequencies of the terms in the comments. In this work, terms with top 3 tf-idf values are identified as the tags of the category. The category tags of a category are presented visually with different font sizes according to their tf-idf values. A tag is presented with a large font size if it is with a high tf-idf value, which indicates that the tag is an important keyword of the category.

For each API usage pattern in a category, an exemplary code snippet will be identified by the following definition:

Definition 5 (Exemplary Code Snippet without a Query): Let *r* be the API usage pattern, and *average-tfidf*($c_{s_i^r}$) be the average of the *tfidf* values of the terms in $c_{s_i^r}$. The exemplary code snippet of *r* without a query is the code snippet that directly follows the comment whose *average-tfidf* value is the largest one among the ones in the set $\{c_{s_1^r}, \dots, c_{s_m^r}\}$.

Table 2. Proposed Metric of measuring distances between two code snippets.

Locations of code snippets *x* and *y*	distance(*x*, *y*)
x and *y* are in the same method, and *y* is in the following code of *x*	$1 - \frac{1}{FisrtLineNo_y - FirstLineNo_x}$
x and *y* are in different methods of a class, or *x* is in the following code of *y*	2
x and *y* are in different classes of a package	3
x and *y* are in different packages of a project	4
x and *y* are in different projects	5

The comment of the exemplary code snippet would consist of popular keywords for the API usage pattern.

2.3 Recommending Follow-up API Usage Patterns based on Distances between Patterns

Once an API usage pattern is selected by a programmer from the search results or a category, a number of API usage patterns will be recommended for writing the follow-up code based on distances between usage patterns.

The distances between API usage patterns are calculated based on the distances between their associated code snippets. The distance between two code snippets is defined in Table 2 in which there are five types of relationships between two code snippets according to their locations.

For the first type, the distance from code snippet x to y is measured as $1-\frac{1}{FirstNo_y - FirstNo_x}$, where $FirstLineNo_x$ and $FirstLineNo_y$ denote the first line numbers of x and y, respectively. The distance from x to y will be calculated as a small value as the first line number of y is close to that of x. For the rest of four types, the distance will be determined based on whether the code snippets are in different methods, classes, packages, or projects.

Based on the distances between code snippets, distances between API usage patterns will be calculated by the following metric:

Definition 6 (Metric of Measuring Distances between API Usage Patterns) Let r and $t \in AP$ be two API usage patterns with code snippets $s_1^r, \ldots, s_n^r \in S_r$ and $s_1^t, \ldots, s_m^t \in S_t$. The distance from r to t is calculated as

$$distance(r,t) = \frac{\sum_{i=1}^{n} distance(s_i^r, t)}{n}, \tag{6}$$

where $distance(s_i^r, t) = \mathbf{min}\{distance(s_i^r, s_1^r), \ldots, distance(s_i^r, s_m^r)\}$.

For each code snippet s_i^r of regularization form r, distances from s_i^r to all the code snippets of regularization form t are calculated, and the minimum distance value is selected as the distance from s_i^r to pattern t. The distance from pattern r to pattern t is calculated as the average of the distances from all the code snippets of regularization form r to t. In the system, distances between any two patterns will be calculated.

Once an API usage pattern $r \in AP$ is selected from the search results or a category by the programmer, the k nearest usage patterns of r will be recommended. In this work, k is set to 5. In addition, for each recommended usage pattern, an exemplary code snippet will also be identified through Definition 5.

3. EXPERIMENTAL EVALUATION

In this section, we present the statistics of the discovered API usage patterns (Section 3.1) and a developed API usage pattern assistant tool (Section 3.2). Additionally, we also evaluated the precision and recall of API Usage patterns categorization (Section 3.3), and the hit rate of API usage patterns recommendation (Section 3.4).

Table 3. Lines of code of the open source projects.

Number of Projects	Min LOC	Max LOC	Average LOC	Std Dev.
10510	33	3,709,737	30,011	102,244

3.1 Discovered API Usage Patterns

In the experiment, we collected 10,510 open source projects from sourceforge.net, eclipse.org and apache.org. The minimum project LOC is 33 and the maximum one is 3,709,737. The average LOC of the projects is 30,011 with standard deviation of 102,044 (see Table 3). In addition, we observe that the lines of comments are highly correlated with project lines of code with Pearson correlation coefficient of 0.942 (See Fig. 4). After parsing the source code of the projects, 2,585,906 code snippets are extracted, and 1,775 API usage patterns are discovered. Fig. 5 shows the distribution of numbers of projects in which an API usage pattern appears. The average number of projects in which a usage pattern appears is 11.8.

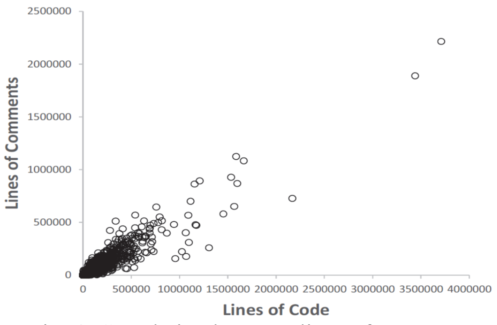


Fig. 4. Correlation between lines of comments and project lines of code.

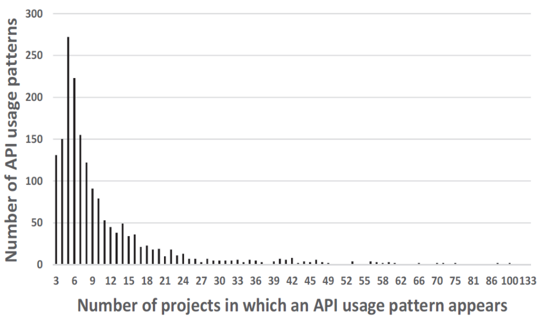


Fig. 5. Distribution of numbers of projects in which an API usage pattern appears.

Table 4. Lines of code of the API usage patterns.

LOC	Number of API Usage Patterns
2	1296
3	357
4	70
5	35
6	10
7	4
8	2
9	1
	Total: 1775

Table 4 shows the distribution of the lines of code of the discovered API usage patterns. A large number of usage patterns are with 2 lines of code, and the average lines of code of a usage pattern is 2.4. In order to better demonstrate the results of this work with a controllable number of API usage patterns, API usage patterns with only 1 line of code are not considered in this experiment.

In order to evaluate how the discovered API usage patterns appear in projects other than the 10,510 projects. Another 474 open source projects are collected and the number of the usage patterns appearing in each project is counted. Fig. 6 shows the correlation between the number of usage patterns and the project lines of code. It is observed that they are highly correlated with Pearson correlation coefficient of 0.85, which means the more lines of code a project has, the more API usage patterns appear.

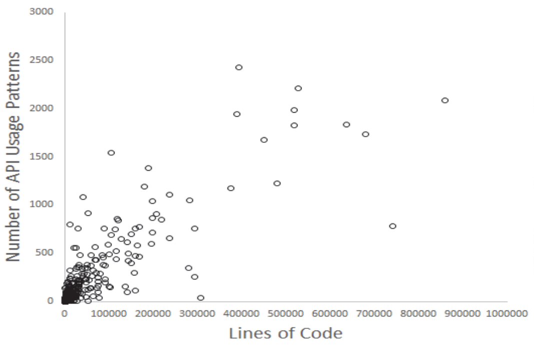


Fig. 6. Correlation between number of API usage patterns and project lines of code.

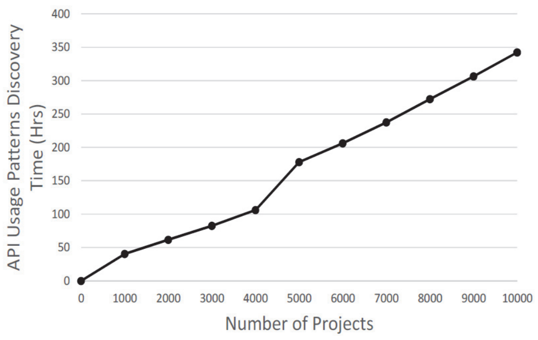


Fig. 7. Performance of discovering API usage patterns.

Fig. 7 shows the performances of discovering API usage patterns with Intel Core i7 3.40 GHz, 16 GB RAM, Windows 7, and MySQL Server v5.6. For example, it takes 40.5 hours and 342.5 hours to discover API usage patterns with 1,000 and 10,000 projects, respectively. With a paired samples *t*-test, the *t*-value for comparing the performances with the ones of 0.034×project_numbers is 0.958 (< 2.262) with degrees-of-freedom=9 and α=0.05, which indicates that there is no significant difference between the performance of our approach and $O(n)$.

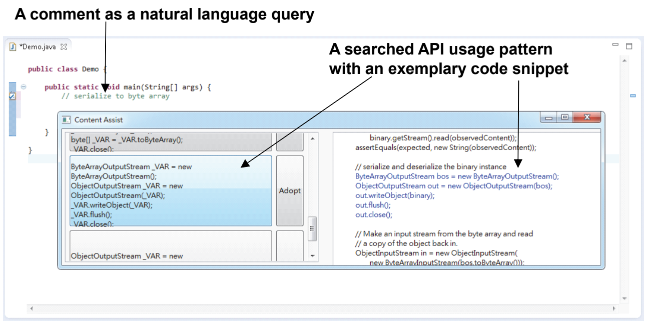
The effectiveness of the API usage patterns discovery is evaluated with two different settings: one is discovering with removing stop and project specific code statements (the proposed approach); and the other is discovering without removing them (the naive approach). Table 5 shows the numbers of API usage patterns discovered. With removing the stop and project specific code statements, 222 additional API usage patterns can be found with 14.3% increase.

Table 5. The effectiveness of the API usage patterns discovery.

API Usage Pattern Discovery Approach	Number of Discovered API Usage Patterns
The Naive Approach (Without removing stop and project specific code statements)	1553
The Propose Approach (With removing stop and project specific code statements)	1775

3.2 An API Usage Pattern Assistant Tool

In this work, we implemented an API usage patterns assistant tool as an Eclipse plugin. Fig. 8 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 addition, a programmer is also enabled to choose an API usage pattern through a category view (see Fig. 9). After a programmer adopts a usage pattern, a list of recommended follow up usage patterns will be presented on the right hand side of the tool (see Fig. 10.).


A comment as a natural language query
A searched API usage pattern with an exemplary code snippet
*Demo.java
Content Assist
Adopt

```
public class Demo {

    public static void main(String[] args) {
        // serialize to byte array

    }
}
```

```
byte[] _VAR = _VAR.toByteArray();
_VAR.close();
```

```
ByteArrayOutputStream _VAR = new
ByteArrayOutputStream();
ObjectOutputStream _VAR = new
ObjectOutputStream(_VAR);
_VAR.writeObject(_VAR);
_VAR.flush();
_VAR.close();
```

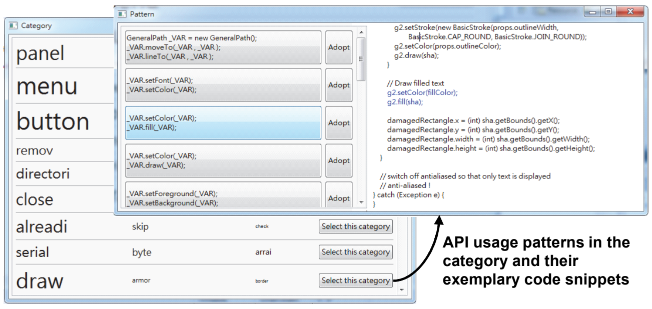
```
ObjectOutputStream _VAR = new
```

```
        binary.getStream().read(observedContent));
    assertEquals(expected, new String(observedContent));

    // serialize and deserialize the binary instance
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(bos);
    out.writeObject(binary);
    out.flush();
    out.close();

    // Make an input stream from the byte array and read
    // a copy of the object back in.
    ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(bos.toByteArray()));
```

Fig. 8. A snapshot of searching API usage patterns



```
GeneralPath _VAR = new GeneralPath();
_VAR.moveTo(_VAR , _VAR );
_VAR.lineTo(_VAR , _VAR );
```

```
_VAR.setFont(_VAR);
_VAR.setColor(_VAR);
```

```
_VAR.setColor(_VAR);
_VAR.fill(_VAR);
```

```
_VAR.setColor(_VAR);
_VAR.draw(_VAR);
```

```
_VAR.setForeground(_VAR);
_VAR.setBackground(_VAR);
```

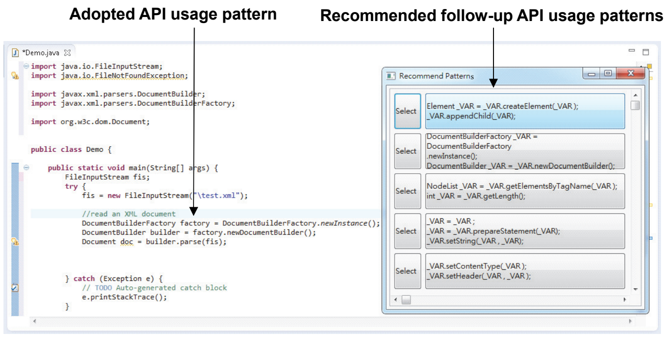
```
        g2.setStroke(new BasicStroke(props.outlineWidth,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
        g2.setColor(props.outlineColor);
        g2.draw(sha);
      }

      // Draw filled text
      g2.setColor(fillColor);
      g2.fill(sha);

      damagedRectangle.x = (int) sha.getBounds().getX();
      damagedRectangle.y = (int) sha.getBounds().getY();
      damagedRectangle.width = (int) sha.getBounds().getWidth();
      damagedRectangle.height = (int) sha.getBounds().getHeight();
    }

    // switch off antialiased so that only text is displayed
    // anti-aliased !
  } catch (Exception e) {
  }
```

Fig. 9. A snapshot of browsing API usage patterns categories.


Adopted API usage pattern
Recommended follow-up API usage patterns
*Demo.java
Recommend Patterns
Select

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;


public class Demo {

    public static void main(String[] args) {
        FileInputStream fis;
        try {
            fis = new FileInputStream("\test.xml");

            //read an XML document
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(fis);



        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

```
Element _VAR = _VAR.createElement(_VAR );
_VAR.appendChild(_VAR);
```

```
DocumentBuilderFactory _VAR =
DocumentBuilderFactory
.newInstance();
DocumentBuilder _VAR = _VAR.newDocumentBuilder();
```

```
NodeList _VAR = _VAR.getElementsByTagName(_VAR );
int _VAR = _VAR.getLength();
```

```
_VAR = _VAR ;
_VAR = _VAR.prepareStatement(_VAR);
_VAR.setString(_VAR , _VAR);
```

```
_VAR.setContentType(_VAR );
_VAR.setHeader(_VAR , _VAR );
```

Fig. 10. A snapshot of recommending follow up API usage patterns.

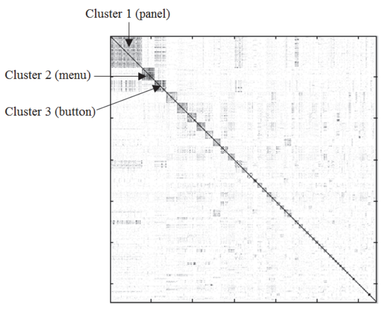


Fig. 11. The correlation matrix of the API usage patterns.

3.3 Precision and Recall of API Usage Patterns Categorization

In the experiment, a cluster is considered as a category if it contains at least 3 API usage patterns. After applying the proposed clustering algorithm to the 1775 usage patterns with threshold *h*=0.72, 644 categories are found.

We visualize the categories by a correlation matrix between the API usage patterns as depicted in Fig. 11. The usage patterns of the same category are put together in the matrix. The color of each cell represents the semantic similarity degree between two API usage patterns. A dark gray cell represents a high semantic similarity degree (high correlation), and a light gray cell represents a low one (low correlation). The matrix clearly shows that the correlations between the API usage patterns of each category are represented as a dark gray square, which means the usage patterns inside the same category are more semantically similar to each other than those outside the category.

Table 6 shows the top 10 categories of the API usage patterns ranked by the number of contained usage patterns. Each category is with three automatically identified category tags. The category tags are represented with different font sizes according to their tf-idf values. The top 3 categories are about the Java user interfaces and are visualized as the three largest dark gray squares in the correlation matrix in Fig. 11.

Table 7 shows top 5 API usage patterns in category 9 with a highlighted keyword "draw". The center usage pattern of the category is about drawing a string on a canvas, and the other four usage patterns are about drawing a shape, drawing a rectangle, drawing a line and setting the stroke of a Graphs2D object, respectively.

In order to evaluate the precision and recall of the categorization results, the top 10 categories are manually reviewed by three Java experts. The precision and recall are calculated by the following definition:

Table 6. Top 10 categories of the API usage patterns.

#	Category tags	Center API usage pattern	Total # of patterns in the cluster	# of relevant patterns in the Cluster	Total # of relevant patterns	Precision	Recall

1	panel pane button	JPanel _VAR = new JPanel();
		_VAR.setLayout(new BorderLayout());
			77	73	88	0.948	0.83

2	menu menubar help	JMenuBar _VAR = new JMenuBar();
		_VAR .setJMenuBar(_VAR);
			31	27	39	0.871	0.692

3	button composite create	_VAR = new Button(_VAR, SWT.PUSH);
		_VAR.setText(_VAR);
			28	24	29	0.857	0.828

4	remove endswith trail	_VAR = _VAR.substring(_VAR , _VAR.length() - _VAR);	26	25	27	0.962	0.926

5	directory exist file	File _VAR = _VAR.getParentFile();
		if (_VAR != _VAR && !_VAR.exists()) {
		_VAR.mkdirs();
		}
			25	25	29	1.0	0.86

6	close stream flush	_VAR.flush();
		_VAR.close();
			22	19	20	0.864	0.95

7	already skip check	if (_VAR.contains(_VAR))
		continue;
			21	15	25	0.714	0.6

8	serialize byte array	ObjectOutputStream _VAR = new ObjectOut-putStream(_VAR);
		_VAR.writeObject(_VAR);
			19	16	17	0.842	0.941

9	draw armor border	_VAR.setColor(_VAR);
		_VAR.drawString(_VAR, _VAR, _VAR);
			18	15	20	0.833	0.75

10	dom document xml	Transformer _VAR = TransformerFacto-ry.newInstance().newTransformer();
		_VAR.setOutputProperty(OutputKeys.INDENT , _VAR);
			17	11	12	0.647	0.917

					Average:	**0.854**	**0.83**

Table 7. An example of the API usage patterns in a category.

Category 9	
API usage pattern	Similarity between the Pattern and the Center Pattern

_VAR.setColor(_VAR);
_VAR.drawString(_VAR, _VAR, _VAR);
	1

_VAR.setColor(_VAR);
_VAR.draw(_VAR);
	0.5

_VAR.setColor(_VAR);
_VAR.drawRect(_VAR , _VAR , _VAR , _VAR);
	0.434

_VAR.setColor(_VAR);
_VAR.drawLine(_VAR , _VAR , _VAR , _VAR);
	0.403

_VAR.setStroke(_VAR);
_VAR.setColor(_VAR);
	0.402

Definition 7 (Precision and Recall of API Usage Patterns Categorization): Given a category *c*, the precision of the categorization result is defined as

$$Precision_c = \frac{|\,Rel_c\,|_\#}{|\,Total_c\,|_\#}, \tag{7}$$

where Rel_c denotes the relevant API usage patterns in cluster *c*, and $Total_c$ denotes the total API usage patterns in cluster *c*. The recall of the categorization result is defined as

$$Recall_c = \frac{|\,Rel_c\,|_\#}{|\,TotalRel_c\,|_\#}, \tag{8}$$

where $TotalRel_c$ denotes the total relevant API usage patterns.

Table 6 shows the values of the precision and recall for each category. The average precision is 85.4% and the average recall is 83%.

3.4 Hit Rate of API Usage Patterns Recommendation

In the experiment, the 5 nearest API usage patterns (k=5) will be recommended for each adopted usage pattern. Table 8 shows the examples of the top recommended follow-up API usage patterns. For the first usage pattern on creating a menu item, the system recommends a usage pattern on creating a menu that can be used to contain a menu item.

Table 8. Examples of the top recommended follow-up API usage patterns.

#	Adopted API usage pattern	Top recommended usage pattern	Exemplary code snippet

1	JMenuItem _VAR = new JMenuItem();
	_VAR.add(_VAR);
	_VAR.setText(_VAR);
		_VAR = new JMenu();
		_VAR.add(_VAR);
		_VAR.setText(_VAR);
		_VAR.setMnemonic(_VAR);
			jMenuProgram = new JMenu();
			jMenuBar.add(jMenuProgram);
			jMenuProgram.setText("Program");
			jMenuProgram.setMnemonic('P');

2	_VAR = new JTextArea();
	_VAR.setEditable(_VAR);
		_VAR.setText(_VAR);
		_VAR.setCaretPosition(_VAR);
			dateField.setText(m_displayTextToVerify);
			dateField.setCaretPosition(0);

3	_VAR.setColor(_VAR);
	_VAR.fillRect(_VAR , _VAR , _VAR , _VAR);
		_VAR.setColor(_VAR);
		_VAR.drawRect(_VAR , _VAR , _VAR , _VAR);
			g.setColor(borderColor);
			g.drawRect(0, 0, width - 1, height - 1);

4	DocumentBuilderFactory _VAR = DocumentBuilderFacto-ry.newInstance();
	DocumentBuilder _VAR = _VAR.newDocumentBuilder();
	Document _VAR = _VAR.parse(_VAR);
		Element _VAR = _VAR.createElement(_VAR);
		_VAR.appendChild(_VAR);
			Element xmlComplexType = docu-ment.createElement ("xsd:complexType");
			xmlElement.appendChild(xmlComplexType);

For the second usage pattern on creating a text area, a usage pattern on setting text and insertion caret position of a text area is recommended. For the third usage pattern on filling a rectangle, a usage pattern on drawing the outline of a rectangle is recommended. For the fourth usage pattern on building an XML document, a usage pattern on appending a child element in an XML document is recommended.

We collected another 20 open source projects for evaluating the hit rate of the API usage patterns recommendations. The project names and their lines of code are depicted in Table 9. The hit rate of API usage patterns recommendation is calculated by the following definition:

Table 9. Hit rates of recommending follow-up API usage patterns.

#	Open Source Projects	LOC	$\|SnipRec\|_\#$	$\|SnipCRec\|_\#$	Hit_5
1	Direct Democracy Portal 1.0-alpha2	4,642,141	7,223	4,074	0.564
2	ZephyrSoft Toolbox	2,896,313	5,623	2,987	0.531
3	Eclipse-GWF prealpha	1,102,342	2,700	1,290	0.478
4	Zaranux prealpha	859,502	1,097	653	0.595
5	Zocalo Prediction Markets 2011.2	779,928	2,196	752	0.342
6	DonorEdge	742,932	337	155	0.460
7	Dresden OCL 3.1.0	679,637	1,100	705	0.641
8	Simply Ajax and Mobile 7.0.1	637,133	768	365	0.475
9	DjAccount prealpha	528,624	1,150	648	0.563
10	Saros – Distributed Party Programming beta	519,762	924	514	0.556
11	DrJava r5756	519,176	1,209	742	0.614
12	EchoPM alpha	481,272	568	291	0.512
13	Durham Metadata Framework for Eclipse-0.2.0	451,798	677	264	0.390
14	Zimbra Collaboration Suite	395,143	1,279	541	0.423
15	DuruBI 0.4.3	389,733	1,130	209	0.185
16	ZTUS Planning	374,752	663	406	0.612
17	Zeidon Java Object Engine	294,912	108	34	0.315
18	BidCodeGenerator Planning	293,388	448	181	0.404
19	Digital Learning Sciences (DLS)-v3.5.2	282,087	435	175	0.402
20	Eclipse Tools for Microsoft Silverlight Planning	281,191	178	34	0.191
				Average Hit Rate:	**0.463**

Definition 8 (Hit Rate of API Usage Patterns Recommendation): Given a project p, let x be a code snippet of an API usage pattern r_x, and y be a code snippet of an API usage pattern r_y. x and y are within the same method of a class in p, and r_y is the follow-up usage pattern of r_x in the method. x is said to be with a correct recommendation if r_y is in the recommendation list for r_x. The hit rate of the recommendation results is defined as

$$Hit_k = \frac{|\, SnipCRec\,|_\#}{|\, SnipRec\,|_\#}, \tag{9}$$

where *SnipCRec* denotes the code snippets with correct recommendations, and SnipRec denotes the total code snippets with recommendations.

The hit rates of the recommendation results for the 20 projects are depicted in Table 9. The average recommendation hit rate is 46.3% with 29813 code snippets, which means the system has approximately half a chance of correctly predicting the first follow-up API usage patterns that were actually used by the programmers with top 5 recommendations.

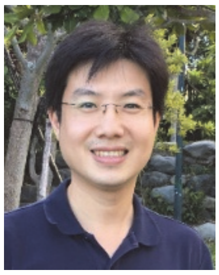
5. CONCLUSION

This paper presents two novel methods for categorizing and recommending API usage patterns through extending our previous work of a comment-driven system for discovering and searching API usage patterns. The system was extended with two key features: first, a programmer is provided with a set of API usage pattern categories that are automatically identified based on a proposed degree centrality-based clustering algorithm; second, a programmer will be recommended a number of follow-up API usage patterns while adopting a usage pattern based on a proposed metric of measuring distances between API usage patterns. We also conduct experiments to validate the proposed approach. The API usage patterns categorization can achieve 85.4% precision rate with 83% recall rate. With top 5 recommendations, API usage patterns recommendation had approximately half a chance of correctly predicting the first follow up API usage patterns that were actually used by the programmers.

REFERENCES

1. Black Duck, Open Hub, https://www.openhub.net/.
2. D. Mandelin, L. Xu, R. BodíK, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 48-61.
3. 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, 2006, pp. 952-970.
4. 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, 2013, pp. 1-27.
5. S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: a search engine for Java using free-form queries," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, 2009, pp. 385-400.
6. H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009, pp. 318-343.
7. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 429-435.
8. J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th International Workshop on Mining Software Repositories*, 2013, pp. 319-328.

9. S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of International Conference on Software Engineering*, 2014, pp. 643-652.
10. W. Janjic, O. Hummel, and C. Atkinson, "Reuse-oriented code recommendation systems," in *Proceedings of Recommendation Systems in Software Engineering*, 2014, pp. 359-386.
11. Git Hub Gist, https://gist.github.com/.
12. Code Recommenders, http://eclipse.org/recommenders/.
13. M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, Vol. 27, 2010, pp. 80-86.
14. T. T. Nguyen, H. A. Nguyen, and N. H. Pham, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009, pp. 383-392.
15. F. Lv, H. Zhang, J Lou, S. Wang, D. Zhang, and J. Zhao, "CodeHow: effective code search based on API understanding and extended boolean model," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 260-270.
16. N. Sahavechaphan and K. Claypool, "XSnippet: mining for sample code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems*, *Languages*, *and Applications*, 2006, pp. 413-430.
17. Krugle Search, http://opensearch.krugle.org/.
18. Merobase Component Finder, http://www.merobase.com/.
19. S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 204-213.
20. Java API, http://docs.oracle.com/javase/7/docs/api/.
21. MSDN Library, http://msdn.microsoft.com/en-us/library.
22. S.-J. Lee, X. Lin, and W.-C. Su, "A comment-driven approach to API usage patterns discovery and search," in *Proceedings of Taiwan Conference on Software Engineering*, 2016, p. 52.
23. G. Erkan and D. R. Radev, "LexRank: graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research*, Vol. 22, 2004, pp. 457-479.
24. D. Jurafsky and J. H. Martin, *Speech and Language Processing*, Pearson Education International, London, 1966, pp. 107-111.
25. A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, Cambridge University Press, UK, 2011.
26. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley, Boston, 1999.



Shin-Jie Lee (李信杰) is an Associate Professor in Computer and Network Center at National Cheng Kung University (NCKU) in Taiwan and holds joint appointments from Department of Computer Science and Information Engineering at NCKU. His current research interests include software engineering and service-oriented computing. He received his Ph.D. degree in Computer Science and Information Engineering from National Central University in Taiwan in 2007.

Wu-Chen Su (蘇武楨) holds a Master of Information Management from National Cheng Kung University, Taiwan (2007). He is currently a research staff at the Department of Clinical Sciences at the University of Kentucky, USA. His main areas of research interest are software engineering, consumer health informatics and clinical informatics.

Chi-En Huang (黃琪恩) is a graduate student in Department of Computer Science and Information Engineering at National Cheng Kung University in Taiwan.

Jie-Lin You (游傑麟) is a graduate student in Department of Computer Science and Information Engineering at National Cheng Kung University in Taiwan.