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

Shin-Jie Lee[1,2], Xavier Lin[2], Wu-Chen Su[3], Hsi-Min Chen[4]

[1] Computer and Network Center, National Cheng Kung University, Taiwan
[2] Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
[3] Department of Clinical Sciences, University of Kentucky, USA
[4] Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan
jielee@mail.ncku.edu.tw, XavierLinX@gmail.com, wuchen_sue@hotmail.com, hmchen@mail.fcu.edu.tw

Abstract

Considerable effort has gone into the discovery of API usage patterns or examples. However, how to enable programmers to search for discovered API usage examples using natural language queries is still a significant research problem. This paper presents an approach, referred to as Codepus, to facilitate the discovery of API usage examples based on mining comments in open source code while permitting searches using natural language queries. The approach includes two key features: API usage patterns as well as multiple keywords and tf-idf values are discovered by mining open source comments and code snippets; and a matchmaking function is devised for searching for API usage examples using natural language queries by aggregating scores related to semantic similarity, correctness, and the number of APIs. In a practical application, the proposed approach discovered 43,721 API usage patterns with 641,591 API usage examples from 15,814 open source projects. Experiment results revealed the following: (1) Codepus reduced the browsing time required for locating API usage examples by 46.5%, compared to the time required when using a web search engine. (2) The precision of Codepus is 91% when using eleven real-world frequently asked questions, which is superior to those of Gists and Open Hub.

Keywords: API usage pattern, Code example, Code search system

1 Introduction

The reuse of existing software is an important part of software development: it saves on time and resources, and improves the quality of the developed software [1-7]. API usage examples are an important source in this endeavour. Currently, there are several sources from which a programmer can retrieve API usage examples, including peers [8], API documentation [9-10], websites with pre-collected code examples [11-12], code search engines [13-15], and web search engines, such as Google Search. When learning programming, many individuals rely on experienced peers to find code examples [8], while others search for code examples in API documentation [9-10]. In [16-17], code examples have been regarded as an important factor in the design of API documentation. A number of websites [11-12] provide sets of pre-collected code examples. However, managing sets of code examples relies heavily on collection effort or content contributed by website users. Code search engines enable programmers to search for code snippets of open source code based on file names, classes, methods, or structures using search methods based on traditional keyword matching [13-15].

Considerable effort has gone into the discovery of API usage patterns or examples [18-25]. However, how to enable programmers to search for discovered API usage patterns or examples using natural language queries is still a significant research problem. A number of approaches are based on the assumption that programmers know precisely what API methods they are going to use [18-19, 21-22, 26] or the types of input/output they require [27-28]. This would enable them to use API method names as queries in the search for relevant API usage patterns or code examples. Unfortunately, this assumption limits the possibility of discovering examples in cases where the programmer is unaware of the relevant method names or input/output types. In this study, we propose an approach, referred to as Codepus, to facilitate the discovery of API usage examples based on mining comments in open source code while permitting searches using natural language queries. The proposed approach includes the following two features:

— *API usage patterns as well as multiple keywords and tf-idf values are discovered by mining open source comments and code snippets*. In the proposed approach, an API usage pattern is defined as an API

*Corresponding Author: Shin-Jie Lee; E-mail: jielee@mail.ncku.edu.tw
DOI: 10.3966/160792642018091905030

usage sequence that appears recurrently in multiple projects. A proposed algorithm enables the extraction of code snippets and the associated comments for use in discovering API usage patterns and the related keywords and term frequency-inverse document frequency (tf-idf) values.

— *A matchmaking function is devised to enable users to search for API usage examples using free-form natural language sentences or phrases*. The matchmaking function aggregates scores related to semantic similarity, correctness, and the number of APIs.

Figure 1 presents an overview of the proposed approach, which can be divided into two parts: (1) discovery of API usage examples and (2) search for API usage examples. In the discovery process, the source code of a number of open source projects is parsed for extracting a large number of code snippets as well as their related comments (see Section 3.1). The extracted code snippets containing project-specific code statements are then removed. The stop code statements commonly used by programmers for logging or debugging (with little or no effect on system functionality) are also filtered out (see Section 3.2). The API usage sequence of each extracted code snippet is then specified. Finally, API usage patterns, examples, related comments, and keywords are identified based on the API usage sequences (see Section 3.3). In the search for API usage examples, users are able to search for API usage examples using free-form queries in natural language. The code snippets related to each query are ranked by a proposed matchmaking function (see Section 3.4).

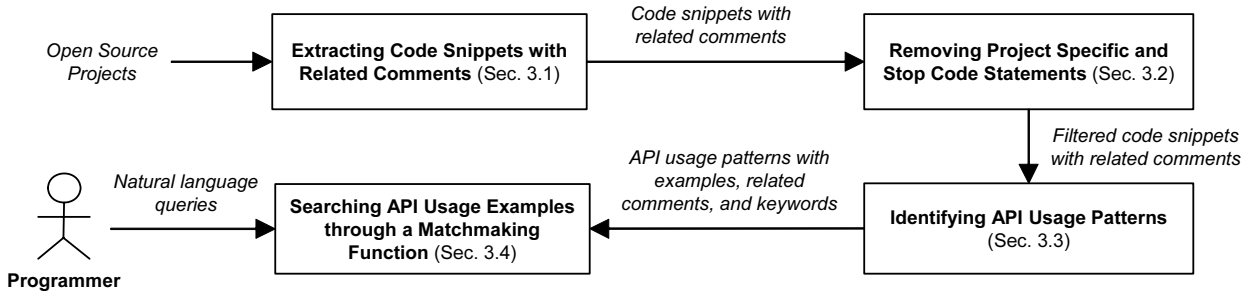


Figure 1. Overview of the proposed approach

The proposed approach is implemented as an Eclipse plugin integrated with a developed web application used in the search for code examples. The efficacy of the proposed approach was evaluated in three experiments: (1) We compared the performance of Codepus with that of Google Search in terms of the time required by programmers to browse search results in order to locate required code snippets. Threats to validity are also discussed in this paper. (2) We compared the precision of Codepus with two existing code search systems. (3) We evaluated the performance of Codepus with regard to computation time in the search for code examples.

The remainder of the paper is organized as follows: Section 2 presents a review of related work. The proposed approach is outlined in Section 3. Section 4 presents the results of experiments to evaluate the proposed approach. Finally, Section 5 summarizes the contributions provided by the proposed approach.

2 Related Work

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

Open Hub [15] 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. In our approach, code snippets are searched through a proposed matchmaking function. In the experimental results, the precision of our approach is statistically significantly superior to the one of Open Hub.

Mandilin et al. [27] 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 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 [29], 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. [22] 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.

Subramanian et al. [30] proposed an iterative, deductive method of linking source code examples to API documentation for increasing the timeliness of the API documentation by providing valuable reference links for source code examples. With an implementation of the method, called Baker, a Stack Overflow post is augmented with links to GitHub source code and Android API documentations, and an Android API documentation is augmented with code examples of Stack Overflow posts.

In [31], Janjic et al. discussed the foundations of software search and reuse, and provided the main characteristics of reuse-oriented code recommendation (ROCR) systems. Moreover, they provided an overview of the architectural organization of a recommender-enhanced IDE to automate a depicted micro-process of software reuse. The micro-process consists of the following key elements: decision to search, description of request, search, recommendation selection, and reuse and maintain. Our tool can be considered as a ROCR system that is specifically designed to assist users in reusing code by natural language queries with support of auto-complementing queries and recommending relevant queries.

GitHub Gists [32] 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. In this work, the experimental results also show that the precision of our approach is statistically significantly superior to the one of Gists.

Code Recommenders [33] is an Eclipse plugin, and one of its features is Snipmatch which provides a way to search for code snippets. Each code snippet is associated with a user-defined metadata including its name, description, keywords, and tags. Code snippets will be searched based on their metadata. Users can share their own code snippets by submitting them to the official repository. All submitted code snippets will be manually reviewed for ensuring the quality and usability. However, currently there are only 128 code snippets available in the default repository. By contrast, in the current implementation of our approach, 43,721 API usage patterns have been discovered through mining 15,814 open source projects.

3 API Usage Patterns Discovery and Search

This section introduces key activities involved in discovering and searching API usage examples: extracting code snippets and related comments (Section 3.1), removing project specific and stop code statements (Section 3.2), identifying API usage patterns (Section 3.3), and searching API usage examples through a proposed matchmaking function (Section 3.4). Implementation details of the proposed approach are also presented (Section 3.5).

3.1 Extracting Code Snippets and Related Comments

In this activity, source code of multiple open source projects are parsed and a large number of code snippets with their related comments are extracted through a proposed algorithm shown in Table 1. The basic concept of the algorithm is to parse each source file in open source projects, extract all comments from the file, and find the following nearest code snippet for each comment. After parsing a source file, the relations between the extracted comments and code snippets will be recorded. The algorithm consists of the following steps.

First (lines 1-4), the source code files, denoted as F, of all projects, denoted as P, are prepared. Two empty sets S and C are declared in order to collect extracted code snippets and the related comments, respectively.

Second (line 5), three functions *cmt*, *file* and *prj* are defined for relating a code snippet with a comment, a file and a project, respectively.

Third (lines 6-7), all comments in the bodies of the methods in every source file $f \in F$ of project $p \in P$ are extracted through parsing the source code. In this work, Eclipse JDT API was used for parsing the Java source files of the open source projects.

Fourth (lines 8-12), for each comment c, the code block s that directly follows c is extracted. A code block is a code snippet of the form *statement* {*statement*}.

Table 1. 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;
3:  let S = Ø;      // to collect code snippets
4:  let C = Ø;              // to collect comments
5:  let cmt: S → C, file: S → F, prj: S → P;
6:  for each source code file f ∈F of project p∈P do
7:        Extract all comments in the bodies of the methods in f;
8:        for each comment c in f do
9:          if there exists a code block s that directly follows c then
10:           let S = S ∪ {s};
11:           let C = C ∪ {c};
12:           let cmt(s)=c, file(s)=f, prj(s)=p;
13:         else
14:           let nextCmt = The start line of the following nearest comment inside the blocks containing c;
15:           let nextBlankLine = The following nearest blank line inside the blocks containing c;
16:           let nextEndOfBlock = The following nearest end line of the block among the blocks containing c;
17:           let s′ be a code snippet in between c and minLineNum(nextCmt, nextBlankLine, nextEndOfBlock);
18:           let S = S ∪ {s′};
19:           let C = C ∪ {c};
20:           let cmt(s′)= c , file(s′)=f, prj(s′)=p;
21:         end if
22:       end for
23: end for
```

For example, in Table 2, lines 5-22 is a method declaration code block, and lines 15-21 is an if-statement code block. In this work, we said that a code block directly follows a comment if the code block follows the comment without any non-blank lines in between them. In the example, only the method declaration code block (lines 5-22 in Table 2) is considered as a directly following code block of a comment (lines 2-4 in Table 2), and it will be extracted together with the related comment.

At last (lines 13-20), if there are no code blocks directly following a comment, code snippets of non-block type will be searched. In order to determine the line number of the end of the following code snippet to be extracted, code lines of the following three types will be identified as boundaries:

1. *The start line of the following nearest comment inside the blocks containing the comment.* For example, for the comment of line 16 (*//Parse data*) in Table 2, line 18 (*//Show data*) is considered as the following nearest comment inside the same block (lines 15-21).

2. *The following nearest blank line inside the blocks containing the comment*. For example, for the comment of line 6 (*//Read data from a given file. The f_name should not be null*), line 14 (a blank line) is considered as the following nearest blank line inside the same block (lines 5-22).

3. *The following nearest end line of the block among the blocks containing the comment.* For example, for the comment of line 18 (*//show data*), there are two blocks, lines 15-21 and lines 5-22, containing the comment, and line 21 is considered as the following nearest end line of the comment.

Table 2. A Source Code Example

```
1:  …
2:  /*
3:  * Show map data from file.
4:  */
5:  public void showMapData(String f_name) throws
    IOException{
6:      //Read data from a given file. The f_name should
        not be null.
7:      File in_file = new File(f_name);
8:      int i = (int) in_file.length();
9:      byte[] data = new byte[i];
10:     FileInputStream fis = new
        FileInputStream(in_file);
11:     fis.read(data);
12:     fis.close();
13:     System.out.println(new String(data));
14:
15:     if(!data.equals("")){
16:         //Parse data
17:         MapData map_data = new
            MapData(data);
18:         //Show data
19:         MapGUI map_gui = new MapGUI();
20:         map_gui.show(map_data);
21:     }
22: }
23: …
```

Among the identified code lines of these three types, the line with the minimal line number is selected as the boundary. The code snippet starting from the next line of the comment to the previous line of the boundary is extracted and is identified to be related with the

comment. For example, code snippets of lines 7-13, line 17 and lines 19-20 will be extracted and related with the comments of line 6, line 16 and line 18, respectively. If there are no code lines of these three types identified, no code snippet will be extracted to be related with the comment.

Because we observed that not all of the sentences in a comment are useful for searching API usage examples, even some have negative impacts, only the first sentence in the comment is retained. Additionally, symbols in the first sentence are also removed. For example, the comment of line 6 will be filtered as *"Read data from a given file"*.

3.2 Removing Project Specific and Stop 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 as a project specific code statement if it consists of invocations of methods that are written in the same project of the code statement. The extracted code snippets containing one or more project specific code statements will be removed.

For example, it is assumed that MapData and MapGUI are classes in the project of the code snippet in Table 2. As line 17 contains an invocation of the constructor of MapData, the code snippet of the line is considered as a project specific code statement and hence will be removed from the set of extracted code snippets. In the same way, the code snippet of lines 19-20 will be removed due to the invocations of the constructor and method show of MapGUI, and the code snippet of lines 5-22 will be removed as well.

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), in line 13 of Table 2 is often used by programmers for logging or debugging and does not contribute to system functionalities in most cases. In this work, 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 statements that satisfy both of the following two criteria are collected:

- The code statement describes an invocation of a method in ignore of its arguments.
- 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 object to be invoked is with the same identifier name in different projects. Table 3 shows the top 5 frequent 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 are removed from the extracted code snippets. For example, line 13 in Table 2 is considered as a stop code statement, and it will be removed from the extracted code snippet of lines 7-13.

Table 3. 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

3.3 Identifying API Usage Patterns

Once the project specific and stop code statements are removed, the API usage sequence of each extracted code snippet will be specified. An API usage sequence is defined as follows.

Definition 1 (API Usage Sequence). Given a code snippet $s \in S$, the API usage sequence q of s is the sequence of the APIs that are sequentially used in s, which is denoted by the function seq(s) = q. Q denotes the set containing the API usage sequences of all code snippets in S.

Table 4 shows an extracted code snippet from the source code example, its related comment, and an identified API usage sequence. There are 5 APIs used in the code snippet. Based on Definition 1, an API usage pattern is defined as follows:

Table 4. An API Usage Sequence Example

Related Comment

1: *Read data from a given file*

Extracted Code Snippet

```
1: File in_file = new File(f_name);
2: int i = (int) in_file.length();
3: byte[] data = new byte[i];
4: FileInputStream fis = new FileInputStream(in_file);
5: fis.read(data);
6: fis.close();
```

API Usage Sequence

```
1: java.io.File.File
2: java.io.File.length
3: java.io.FileInputStream.FileInputStream
4: java.io.FileInputStream.read
5: java.io.FileInputStream.close
```

Definition 2 (API Usage Pattern). *An API usage pattern r is an API usage sequence that recurrently appears in multiple projects. All API usage patterns are denoted as a set* $AP = \{r | r \in Q;$ *and* $| P_r | \geq k\}$, *where* $P_r = \{p | p \in P;\ p = prj(s);$ *and* $s \in S_r\}$ *is the set of projects in which the pattern r appears.* $S_r = \{s | s \in S;$ *and* $seq(s) = r\}$ *denotes the set of code snippets related to r.*

s is called as an API usage example of r. k is the minimum number of projects in which an API usage pattern appears.

In order to better discover API usage patterns that are frequently used by various programmers, an API usage sequence is identified as an API usage pattern if it recurrently appears in multiple projects (*k* is set to 2 by default). Because a code snippet may be copied and pasted multiple times in the source code of the same project by a programmer, duplicate appearances of an API usage pattern in the same project is not considered in the usage pattern identifications.

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

1. *Generate a document d_r related to an API usage pattern r by aggregating the comments of the code snippets related to r*. The document is formally defined as d_r ={c|c∈C; c=cmt(s); and s∈ S_r }, and the documents for all of the API usage patterns are defined as D= $\{d_r \mid r \in AP\}$.

2. *Decompose compound words in document d_r into simple words based on a camel case mapping table.* For example, the compound word "InputStream" will be mapped to two simple words "Input" and "Stream". In order to automatically split a compound word, the mapping table is prepared through the following steps. Firstly, all camel case compound words in all comments of *D* are extracted. Secondly, if a camel case compound word appears in multiple (more than a threshold, and 3 by default) comments, it will be decomposed into several simple words based on commonly used camel case rules. At last, the compound word together with its related simple words will be added into the mapping table.

3. *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 [34]. 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".

4. Calculate the tf-idf value of each word in d_r . tf-idf (term frequency-inverse document frequency) formula 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 :

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

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:

$$idf_{t^r} = \log(\frac{|D|}{1+|d \in D : t^r \in d|}) \quad \textbf{(2)}$$

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

$$tf\,id_{t^r} = tf_{t^r} \times id_{t^r} \quad \textbf{(3)}$$

5. *Identify the terms of top p tf-idf values in d_r as the keywords of the API usage pattern r*. The keywords are denoted as a set $K_r = \{t \mid t \in d_r; t \text{ is with a top p tf - idf value}\}$. In this work, p is set to 8.

Figure 2 shows the relationships between API usage patterns, code snippets, comments, and keywords. r_1 is an API usage pattern. Code snippets $s_1^{r_1}, ..., s_m^{r_1} \in S_{r_1}$ are examples of r_1 . $c_{s_1}^{r_1}, ..., c_{sm}^{r_1}$ are comments that are related to $s_1^{r_1}, ..., s_m^{r_1}$, respectively. The comments are aggregated as a document $d_{r_1} \in D$ from which keywords $t_1^{r_1}, ..., t_p^{r_1}$ with tf-idf values are identified. Section 3.4 will introduce how to rank a code snippet based on the comments and API usage pattern through a proposed matchmaking function.

3.4 Searching API Usage Examples through a Matchmaking Function

For a given query, each code snippet will be ranked based on a proposed matchmaking function that aggregates the scores on semantic similarity, correctness, and API numbers (see Figure 2). The first type of score is the semantic similarity between a query and a code snippet. The score is calculated as follows.

Definition 3 (Score on Semantic Similarity). *Let A and B be two sets of terms with tf-idf values. The cosine similarity between A and B is calculated as*

$$cosine(A,B) = \frac{\sum_{t \in A \cup B} tf\,idf_{t,A} \times tf\,idf_{t,B} \times syn(t)}{\sqrt{\sum_{t \in A^\circ} tf\,idf_{t,A}{}^2} \times t\sqrt{\sum_{t \in B} tf\,idf_{t,B}{}^2}}. \quad \textbf{(4)}$$

Let q be a natural language query consists of a set of terms and s_1 be a code snippet. The score on semantic similarity is calculated as

$$SSS(q, s_1) = cosine(T_q, T_{s_1}), \quad \textbf{(5)}$$

where T_q denotes the terms in q, and T_{s_1} denotes the terms in the comment related to s_1.

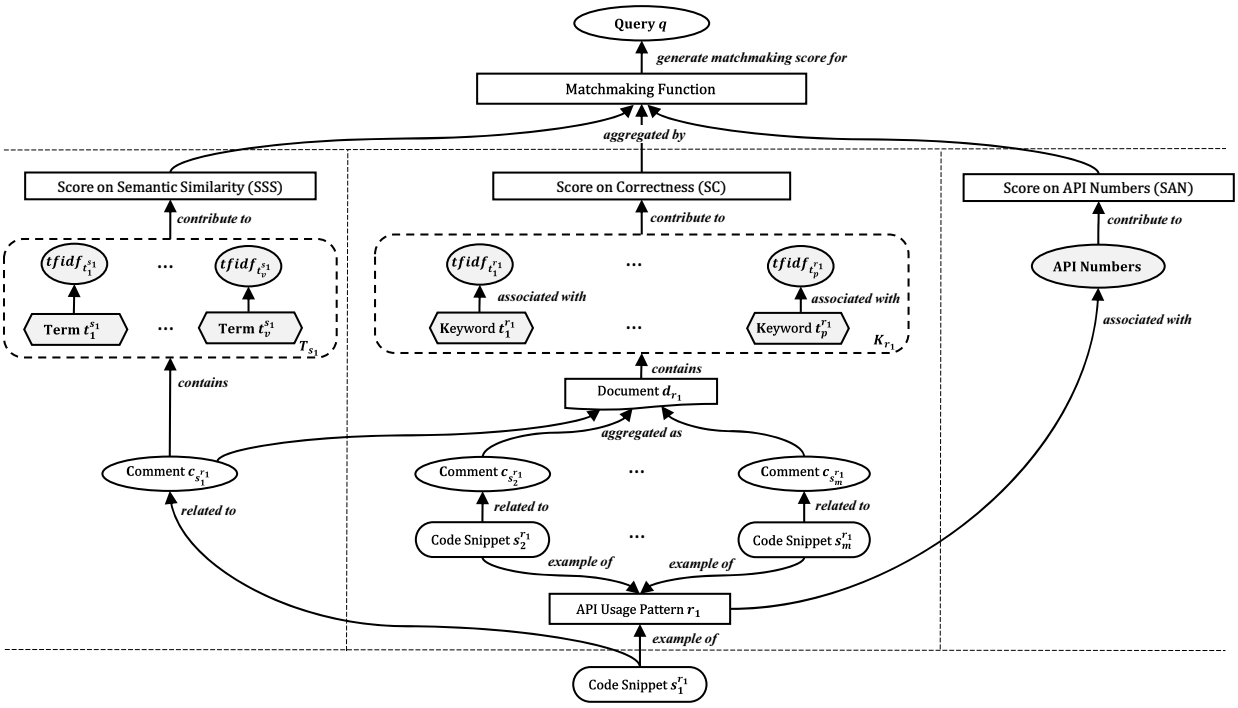


Figure 2. Relationships between API usage patterns, code snippets, comments, and keywords

Because some synonyms may appear in the terms of a query and a comment, an external system, WordNet, was used for detecting the synonyms. In Equation 4, a function $syn(t)$ is introduced to adjust the weighting of synonyms. If a term t appears both in A and B, the value of $syn(t)$ will be 1. If a term t in A is a synonym of a term (also denoted as t in Equation 4) in B, the value of $syn(t)$ will be a real number in between 0 and 1 (the default value is 0.8).

The term frequencies (tf values) of terms in query T_q and comment T_{s_1} are calculated as the raw frequencies of the terms in T_q and T_{s_1} divided by the sums of the raw frequencies of all terms in T_q and T_{s_1}, respectively. The inverse document frequencies (idf values) of the terms are calculated by Equation 2. An SSS value ranges from 0 to 1.

The second type of score is on the correctness of the semantic similarity score. Although every extracted code snippet is with a related comment, some relations between comments and code snippets may be incorrect. For example, a comment "Need to recheck here!" is directly followed by a code snippet on file copy, and a relation built between the comment and the code snippet would result in getting an inappropriate semantic similarity score for matching the code snippet with a query. Therefore, the second type of score is devised to reflect the correctness of the semantic similarity score and is calculated as follows.

Definition 4 (Score on Correctness). *Let q be a natural language query consisting of a set of terms, and r_1 be an API usage pattern related to a code snippet s_1. The score on correctness is calculated as*

$$SC(q, s_1) = cosine(T_q, T_{k_{r_1}}), \quad \textbf{(6)}$$

where T_q denotes the terms in q, and $T_{k_{r_1}}$ denotes the keywords of r_1.

Because the keywords of an API usage pattern are the top important words related to the pattern, some of them may frequently appear in the comments related to the code snippets with the API usage pattern. If most of the terms of the query do not appear in the keywords of the API usage pattern of a code snippet, the score on correctness would get a lower value. An SC value ranges from 0 to 1.

The third type of score is API numbers. If two code snippets are with the same scores on semantic similarity and correctness, the system would like to recommend the user the code snippet of larger code volume for reuse than the smaller one.

Definition 5 (Score on API Numbers). *Let q be a natural language query consisting of a set of terms and r_1 be an API usage pattern related to a code snippet s_1. The score on API numbers is calculated as*

$$SAN(q, s_1) = \begin{cases} \dfrac{\| r_1 \|}{API\_NUM\_BOUND} & if \| r_1 \| < API\_NUM\_BOUND \\ 1 & else \end{cases} \quad \textbf{(7)}$$

where API_NUM_BOUND is a constant, and $\| r_1 \|$ denotes the number of the APIs in r_1

In this work, API_NUM_BOUND is set to 10. For example, if there exists another code snippet with only one line containing a constructor invocation, FileInputStream fis = new FileInputStream(in_file),

and its related comment is the same as the comment in the example of Table 4, the system will give the code snippet a lower SAN value 0.1 ($=\frac{1}{10}$) and a higher one 0.5 ($=\frac{5}{10}$) for the code snippet in Table 4. An SAN value ranges from 0 to 1. Based on the three types of scores, a matchmaking function is defined as follows.

Definition 6 (Matchmaking Function). *Let q be a natural language query consisting of a set of terms and r_1 be an API usage pattern related to a code snippet s_1. An overall score for s_1 is calculated by the following matchmaking function:*

$$matchmaking(q, s_1) = w_1 \times SSS(q, s_1) + w_2 \times SC(q, s_1) + w_3 \times SAN(q, s_1), \quad \textbf{(8)}$$

where w_1, w_2 and w_3 are weights and $w_1 + w_2 + w_3 = 1$.

The matchmaking function aggregates the scores of these three types and returns a value ranges from 0 to 1. The default values of w_1, w_2 and w_3 are 0.4, 0.4 and 0.2, respectively.

3.5 Implementation Details

For discovering API usage patterns and examples, we collected 15,814 open source projects from SourceForge, Eclipse.org, Apache.org, and Google Code. We developed a web crawler to obtain the repository information of all projects listed in the four web sites, and then automatically downloaded the latest versions of the projects data in October 2014 through Git, Subversion, CVS and Hg Client libraries. When a project repository was not available or could not be downloaded, the project was skipped. Once the projects data were downloaded, all of the files in the data were scanned, and only Java source files were parsed. After parsing the Java source files, 5,141,772 code snippets with comments were extracted, and 43,721 API usage patterns with 641,591 API usage examples have been discovered. Table 5 shows the numbers of API usage patterns categorized by number of APIs in a pattern. A large number of API usage patterns have a length of 2 APIs, and the average number of APIs used in a pattern is 2.8.

Table 5. Numbers of Mined API Usage Patterns

Number of APIs in a Pattern	1	2	3	4	5	6	7	8	9	10	11-63	Total
Number of Patterns	6,899	16,593	10,363	5,089	2,368	1,119	603	252	145	66	224	43,721

For searching API usage examples of the discovered usage patterns, a system implementing the Codepus approach was developed. Figure 3 shows the snapshots of the Codepus Eclipse plugin. For example, when a user writes a comment "//read a file" in the source code editor, the user can press "Alt+d" to enable searching for code examples. The plugin will extract the sentence in the comment as a query with removals of special characters and send a request with the query to the web application hosted in the server side through a default Eclipse built-in browser. The search results will be shown below the source code editor in a split panel. The search results contain a list of ranked code snippets with their related comments and a set of recommended relevant comments for further search.

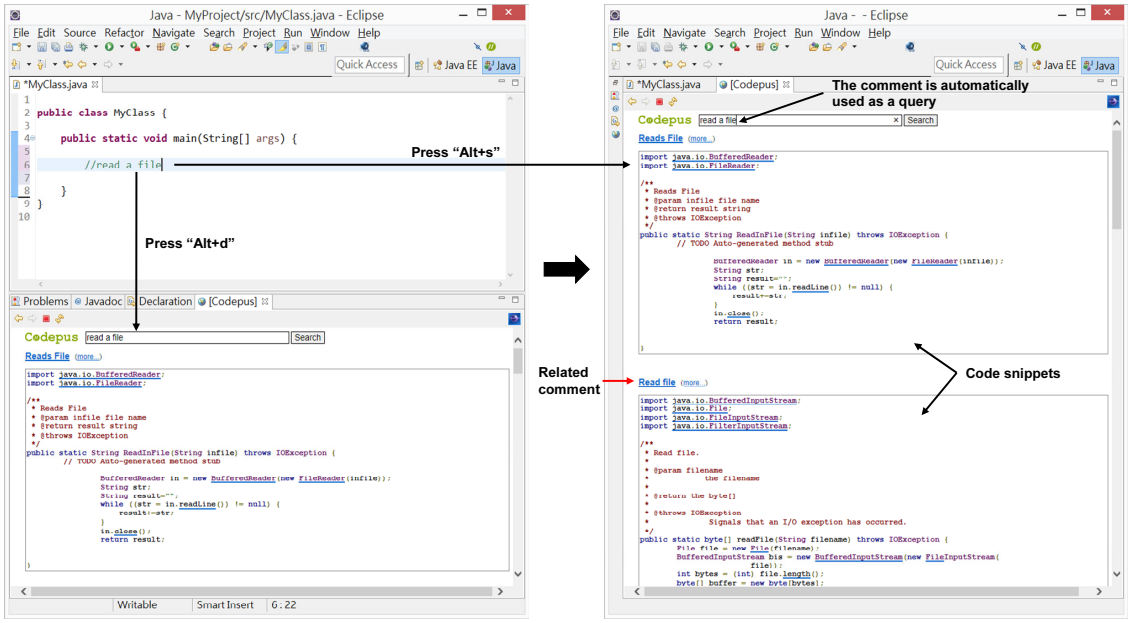


Figure 3. Snapshots of the Codepus Eclipse plugin for searching code examples

4 Experimental Evaluation

This section presents the experimental evaluations of the proposed approach. We designed the experiments to answer the following research questions:

- RQ1: What is the performance of Codepus in terms of the time required by programmers to browse search results in order to locate the required code snippets?
- RQ2: What is the quality of the search results returned by Codepus in terms of precision?
- RQ3: What is the performance of Codepus with regard to computation time on searching code snippets?

4.1 Browsing Time: A Comparison of Codepus and a Web Search Engine

To answer RQ1, we conducted an experiment to evaluate the performance of Codepus in terms of the amount of time programmers spend on browsing search results to locate the code snippets to be adopted.

4.1.1 Design of Experiment

In the experiment, 11 real world Java questions are selected from a question and answer web site, Stack Overflow.

Table 6 shows the selected questions. Each question is then formally modeled as a test problem in Java (see an example in Table 7). Each test problem has a main method and a *problemMethod* method. problemMethod is the method that needs to be implemented to solve the question, and the types of the input and output of problemMethod are determined based on the code shown in the answers of the question from Stack Overflow. In the main method, several comments are added to describe the problem. Basically, the comments describe the input and output of problemMethod. Because the keywords used in the queries for solving the problems would largely influence the quality of search results, we prevented using the keywords appearing in the web pages of the questions except proper nouns.

Table 6. The Selected Top Frequently Asked Questions in Stack Overflow

Q1.	Read/convert an InputStream to a String
Q2.	How to get IP address of current machine using Java
Q3.	Reverse a string in Java
Q4.	File to byte[] in Java
Q5.	How do I remove repeated elements from ArrayList?
Q6.	How to append text to an existing file in Java
Q7.	Renaming a file using Java
Q8.	Convert InputStream to byte array in Java
Q9.	Convert from byte array to hex string in java
Q10.	How do you Programmatically Download a Webpage in Java
Q11.	How can I increment a date by one day in Java?

Table 7. The Test Problem for Question 1 (Q1)

```
1:    import java.io.FileInputStream;
2:    import java.io.InputStream;
3:    public class Problem01 {
4:      static String problemMethod(InputStream inputStream) throws Exception {
5:        // Implement the method
6:        return null;
7:          }
8:  public static void main(String[] args) throws Exception {
9:    // The content of the file "C:/testfiles/file01.txt" is "Hello World 01".
10:    InputStream inputStream = new FileInputStream("C:/testfiles/file01.txt");
11:
12:    // The expected result is "Hello World 01".
13:    String result = problemMethod(inputStream);
14:
15:    System.out.println(result);
16:    /*
17:      The output in the console should be:
18:      Hello World 01
19:    */
20:  }
21:}
```

In the experiment, 20 volunteers with 1-6 years of experience in Java programming were involved and acted as real users of Codepus and Google. They were randomly separated into two groups, says Codepus Group and Google Group, where the users in Codepus Group used Eclipse IDE with Codepus to solve the test problems, while the users in Google Group used Eclipse IDE with Google Search (www.google.com) to solve the same test problems. The experiment is conducted with the following three steps: First, each user reads the guideline document of the experiment. Second, each user solves the pretest problems to get familiar with the style of test problems. At last, each user starts to solve the test problems.

4.1.2 Experiment Results

The section presents the experiment results and elaborates strengths and weaknesses of Codepus. In the experiment, the users gave a number of queries to search for code snippets by Codepus or Google, and the queries can be grouped into two types (see Definition 7).

Definition 7 (Types of Queries). *A query is defined as a Type I query (query with adoption)* q^{T_1} *if the user who gave the query copied a code snippet shown in the search result returned by Codepus or Google in response to the query and pasted it into a test problem; otherwise, the query is defined as a Type II query (query without adoption)* q^{T_2}.

Figure 4 shows a scenario that a user used Codepus to solve a test problem. At first, the user input a query *q1* to search for code snippets, and the search result page was shown at 00:05 ($\alpha_1^{q_1}$). The user started browsing the search result, and then switched the window to the code editor at 00:17 ($\beta_1^{q_1}$) to view the test problem again. At 01:28 ($\alpha_2^{q_1}$), the user switched the window back to the search result page again and continued browsing the search result. Thereafter, the user selected and copied a code snippet. At 01:35 ($\beta_2^{q_1}$), the user switched the window to the code editor again, and then pasted (adopted) the code snippet into the test problem. The user then used the code snippet to solve the test problem. As a result, query *q1* is considered as a Type I query (query with adoption).

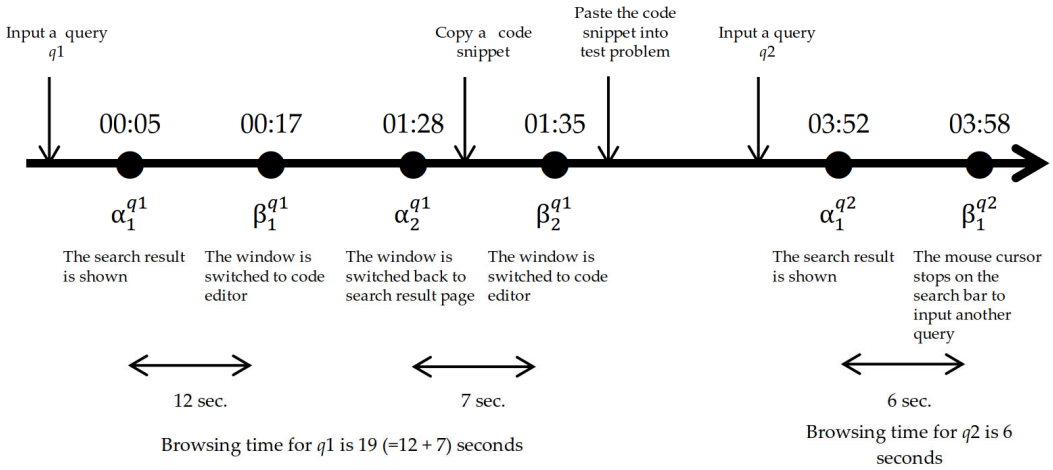


Figure 4. A scenario that contains a Type I query (q1) and a Type II query (q2)

In order to finish the test problem, the user input another query *q2* to search for code snippets, and the search result page was shown at 03:52 ($\alpha_1^{q_2}$). After the user browsed the search result for nearly 6 seconds, the user moved the mouse cursor to the search bar and stopped on it at 03:58 ($\beta_1^{q_2}$), and then typed another query to search for code snippets.

In this scenario, *q1* and *q2* are Type I query and the Type II query, respectively. Furthermore, the time spent on browsing the search result in response to a query is formally defined in Definition 8.

Definition 8 (Browsing Time for a Query). *Given a query q, a time point* α^q *denotes the time at which a search result is returned or the time at which the current window is switched from a non-search result page to a search result page, and a time point* β^q *denotes the time at which the user (a) switches the current window from a search result page to a non-search result page, or (b) moves and stops the mouse cursor on the search bar to type another query. The time* t_q *spent on browsing the search result in response to the query q is calculated as*

$$t_q = \sum_{i=1}^{n} (\beta_1^q - \alpha_1^q). \quad \textbf{(9)}$$

In this place, a search result page returned from Google can be one of the web pages originally linked from the returned web pages list. For example, if a user clicked a link in a web pages list returned from Google, the web page of the link was also viewed as a search result page. In the scenario of Figure 4, the browsing time for *q1* is the sum of $\beta_1^{q_1} - \alpha_1^{q_1}$ (00:05~00:17) and

$\beta_2^{q_1} - \alpha_2^{q_1}$ (01:28~01:35), that is, 19 seconds; and the browsing time for *q2* is $\beta_1^{q_2} - \alpha_1^{q_2}$ (03:52~03:58), that is, 6 seconds. Based on Definition 8, the time spent on browsing search results in response to Type I and Type II queries when a user was solving a test problem is defined as Definition 9.

Definition 9 (Browsing Time in Solving a Test Problem by a User). *Let* $p_i(1 \le i \le 11)$ *be a test problem,* $u_j(1 \le j \le 10)$ *be a user,* $Q_{p_i,u_j}^{T_1} = \{q_1^{T_1}, q_2^{T_1}, ..., q_n^{T_1}\}$ *and* $Q_{p_i,u_j}^{T_2} = \{q_1^{T_2}, q_2^{T_2}, ..., q_m^{T_2}\}$ *be the set of the Type I queries and Type II queries given by user* u_j *when solving test problem* p_i*, respectively. The time spent on browsing the Type I queries and the Type II queries when solving the test problem* p_i *by user* u_j *are calculated as* $t_{Q_{p_i,u_j}^{T_1}} = t_{q_1^{T_1}} + t_{q_2^{T_1}} + \cdots + t_{q_n^{T_1}}$ *and* $t_{Q_{p_i,u_j}^{T_2}} = t_{q_1^{T_2}} + t_{q_2^{T_2}} + \cdots + t_{q_m^{T_2}}$*, respectively.*

There is a need to know the difference between the time the two groups spent on browsing the search results of Type I queries as well as Type II queries.

Definition 10 (Average Browsing Time in Solving a Test Problem by a User). *For each problem* p_i*, the average time a user spent on browsing search results in solving the problem is calculated as*

$$\frac{\sum_{j=1}^{10}(t_{Q_{p_i,u_j}^{T_1}} + t_{Q_{p_i,u_j}^{T_2}})}{\| U_{p_i} \|}, \quad \textbf{(10)}$$

where U_{p_i} *is the set of users who gave at least one query when solving the problem* p_i.

For example, in Table 8, the sum of the browsing time in solving the test problem p_i by all Codepus users is $\sum_{j=1}^{10}(t_{Q_{p_i,u_j}^{T_1}} + t_{Q_{p_i,u_j}^{T_2}}) = 13+156+18+18+43+39+10+26+47+13 = 383$, and there are 9 users who gave at least one query when solving p_i. Therefore, the average browsing time in solving test problem p_i by a Codepus user is $\frac{383}{9} = 42.6$ seconds.

Table 8. Average Browsing Time in Solving Each Test Problem by a User (in seconds)

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	Average
Codepus Group	42.6	18.3	28	55.7	142	49.2	66.4	29.5	48.3	82.8	64	57.0
Google Group	81.8	73.3	34.8	107.9	81.8	58.5	46.8	105.1	163.8	75.2	66.1	81.4

As shown in Table 8, the average browsing time of Codepus group are less than the ones of Google group for 8 test problems out of 11, but are more than the ones of Google groups for 3 test problems. The average values of the browsing time in solving a problem by a Codepus user and by a Google user are 57.0 and 81.4 seconds, respectively.

In summary, the experiment results show that the chance of actually adopting code snippets by a user for a search with Codepus is not significantly different from the one with Google, but the average browsing time a Codepus user spent on locating the code snippets to be adopted for a search is 46.5% less than the time a Google user spent. In addition, the average time a Codepus user spent on browsing search results in solving a problem are less than those of a Google user in the most cases (8 out of 11 test problems).

4.1.3 Discussion

This section discusses the limitations of the current implementation of Codepus. The limitations can be best explored through investigating the reasons why the average browsing time of Codepus group in solving test problems p_5, p_7 and p_{10} are more than those of Google group. We go into details on the users' behavior in solving the three test problems, and provide several insights of the limitations of the current implementation of Codepus.

For problem p_5, the corresponding question in Stack Overflow is "How do I remove repeated elements from ArrayList?". The Codepus users who input more than four Type II queries are u_5, u_7 and u_9. In what follows, the queries given for the problem by the three users are discussed.

The Type II queries given by user u_5 are "bucket sort", "list have", "list check", "list elements" and "list elements count", which are not the phrases commonly used in the queries by the other users, such as "remove repeated" or "delete duplicated". Therefore, Codepus is not likely to return the results that can help the user solve this problem.

The Type II queries given by user u_7 are "remove the same element from list", "no repucated element in List", "List element", "convert List to set" and "set to list". For the first query, although the semantics of the terms "the same" and "duplicate" are similar, the former one is not included in the synonym database (WordNet) used in Codepus system. The word "repucated" in the second query can be considered as a misspelling of "replicated", but currently the implementation of Codepus does not support spell checking. Because the user could not get satisfactory

results with the first two queries, the user then attempted to search with some other queries and wasted much time on them. Therefore, if Codepus can return better results for the first two queries, the user may be able to spend less browsing time for this problem.

User u_{10} input the following queries: "delete repeated elements from list", "return unrepeated elements from list", "return unrepeated elements in list", "remove repeated elements in list", "remove repeated elements", "remove repeated string" and "remove duplicated string". However, the code snippets returned by Codepus for all of the queries were not adopted by the user. After that the user input a query "remove duplicate from list" and adopted a code snippet from the result. Although the terms "delete" and "repeated" are the synonyms of "remove" and "duplicated", respectively, the appropriate code snippets are not ranked in the top 10 list with the setting of the default value 0.8 for weighting synonyms in calculating cosine similarities.

For problem p_7, the corresponding question is "Renaming a file using Java", and 6 users used the phrase "change file name" in the queries to search for code snippets. Although the phrases "rename file" and "change file name" have the same meaning, the latter one does not in the comments of the code snippets related to the problem, and the two phrases are not contained in the synonym database.

For problem p_{10}, the corresponding question is "How do you Programmatically Download a Webpage in Java". Five users used the term "html" in the queries. Because the term "html" does not appear in the comments of the code snippets related to the problem, the users had to spend more time on searching code snippets with other words, such as "http" and "url".

In summary, the current implementation of Codepus can be further improved with the following directions: (a) mining more open source projects to collect more terms related to code snippets; (b) extending the synonym database to include more synonyms, and optimizing the parameters for weighting synonyms; and (c) providing spell checking feature.

4.1.4 Threats to Validity

In this section, we discuss the potential threats that may impact the validity of the experiment results.

Internal Validity. In the context of the evaluation of the performance with regard to browsing time, the main threat to internal validity is the difference between the degrees of Java programming proficiency of the Codepus and Google users. A programmer who is proficient in Java programming may spend less time on browsing code snippets, and therefore it is important to divide the users into two groups with approximately equal degrees of proficiency in Java programming for conducting the experiment. In order to minimize the threat, we randomly separated 20 users into two groups and conducted an examination to evaluate the proficiency degree of each user. The questions of the examination are selected from Oracle Certified Professional Java SE7 Programmer Exams (OCJP) quizzes. With an independent samples t-test, there is no significant difference between the average scores of the two groups.

External Validity. The main threats to external validity of the performance evaluation with regard to browsing time are problems selection and formulation. The first threat is that the selected problems for the experiment might not have been the representative ones in realistic development. We mitigated this threat by selecting 11 top frequently asked real world Java questions from a popular question and answer web site through a systematic selection process, and designing the corresponding test problems in Java. The second threat is that the terms appearing in the test problems might have influenced the users on inputting the words of queries. In practices, the vocabulary programmers can use for constructing queries is open ended. This threat was minimized by formulating each test problem as a Java class with a method to be implemented by the users and preventing using the keywords appearing in the web page of the corresponding question except proper nouns.

4.2 Precision: A Comparison of Codepus and Code Search Systems

In order to answer RQ2, the precisions of three code search systems, Codepus, Open Hub and GitHub Gists, were evaluated and compared. In the experiment, the 11 selected top frequently asked questions in Stack Overflow (listed in Table 6) were used as queries to search code snippets with the three code search systems. For each query, every top one code snippet returned by each code search system was collected. Because Code Recommenders did not return any code snippets for the 11 queries, it was not considered in the experiment of precision evaluation.

Once all of the returns were collected, three experts with over 15 years of experience in Java programming were invited to review the returned code snippets. While an expert reviewed a returned code snippet for a query, the expert had to give a rating for the relevance between the code snippet and the query. A rating is one of the following two values:

- *Relevant: The code snippet is relevant to the query.*
- *Irrelevant: The code snippet is irrelevant to the query.*

The precision of the search result for a query is calculated by the following definition:

Definition 11 (Precision of Search Results). *Given a natural language query q, the precision of the search result is calculated as*

$$Precision_n = \frac{\| Rel \|}{\| Ret \|}, \quad \textbf{(11)}$$

where Ret denotes top n returned code snippets, Rel denotes the code snippets that belong to Ret and are relevant to q. A code snippet is determined to be relevant to a query if all of the experts give ratings of relevant.

Table 9 shows the evaluation results of the precisions of the three code search systems with respective to their returned top-1 results. The average precisions of the three systems for the 11 queries were calculated as 9%, 45% and 91%, respectively. The t-values for comparing the average precisions of Codepus and Open Hub and for comparing the those of Codepus and Gists are 6.708 and 2.887, respectively. Both t-values are greater than 2.228 with 10 degrees of freedom and 95% confidence level, which indicates that the differences between the average precisions of Codepus and Open Hub, and between those of Codepus and Gists are significant. As there may be more than one relevant results in Open Hub and Gists, and the results cannot be directly accessed, the recalls were not measured in the experiment.

Table 9. Comparison of the Precisions of Open Hub, Gists, and Codepus

#	Query	Open Hub			Gists			Codepus		
		$\| Ret \|$	$\| Rel \|$	$Precision_1$	$\| Ret \|$	$\| Rel \|$	$Precision_1$	$\| Ret \|$	$\| Rel \|$	$Precision_1$
1	Read/convert an InputStream to a String	1	0	0.00	1	1	1.00	1	1	1.00
2	How to get IP address of current machine using Java	1	0	0.00	1	0	0.00	1	1	1.00
3	Reverse a string in Java	1	0	0.00	1	1	1.00	1	1	1.00
4	File to byte[] in Java	1	0	0.00	1	0	0.00	1	1	1.00
5	How do I remove repeated elements from ArrayList?	1	0	0.00	1	0	0.00	1	0	0.00
6	How to append text to an existing file in Java	1	0	0.00	1	0	0.00	1	1	1.00
7	Renaming a file using Java	1	0	0.00	1	1	1.00	1	1	1.00
8	Convert InputStream to byte array in Java	1	0	0.00	1	1	1.00	1	1	1.00
9	Convert from byte array to hex string in java	1	1	1.00	1	1	1.00	1	1	1.00
10	How do you Programmatically Download a Webpage in Java	1	0	0.00	1	0	0.00	1	1	1.00
11	How can I increment a date by one day in Java?	1	0	0.00	1	0	0.00	1	1	1.00
	Average Precision:			0.09			0.45			0.91

4.3 Computation Time

This section answers RQ3 by reporting the computation time of Codepus tool for searching code snippets. In the system, the turnaround time of a request can be broken down into three portions: time of transmitting the query string to the server, computation time of processing the request, and time of transmitting the search results back to the client. Because the sizes of a request string and a search results page would not be large in a normal case, we focused more on optimizing the processing of a request. The implementation of Codepus tool consists of the following two optimizations.

Optimization 1. Most of the information for matchmakings and recommendations are cached in memory. The cached data include: keywords with tf-idf values of each API usage pattern, terms with tf-idf values, degree centrality of each extracted comment, and API numbers of each API usage pattern. Additionally, the relations between API usage patterns and code snippet identifiers, and the relations between code snippets and their related terms are also cached and indexed using Java Hashtable API.

Optimization 2. For a query, all *SC* values (score on correctness) for all API usage patterns are computed before computing *SSS* (score on semantics similarity) and *SAN* (score on API numbers) values. If the *SC* value of an API usage pattern is computed as 0, the code snippets related to the pattern will be ignored and the remaining computations for *SSS* and *SAN* values are omitted. Hence, not all of the *SSS* and *SAN* values of all code snippets should be computed, and only a relatively small portion whose *SS* values are greater than 0 needs be calculated.

Table 10 shows the computation time of searching code snippets for the 11 selected queries. The average computation time is 1.546 seconds with standard deviation of 0.843.

Table 10. Computation Time of Searching Code Snippets for the 11 Queries (In Seconds)

Query#	1	2	3	4	5	6	7	8	9	10	11
Computation Time	2.125	1.156	1.093	1.250	3.109	2.063	0.954	1.719	2.547	0.594	0.391
Average time: 1.546 sec											

5 Conclusion

This paper proposes an approach referred to as Codepus for the discovery of API usage examples based on mining comments in open source code using natural language queries. The approach was implemented as an Eclipse plugin tool integrated with a developed code search web application. In a practical application, the proposed approach discovered 43,721 API usage patterns with 641,591 API usage examples from 15,814 open source projects. Experiment results revealed the following: (1) Codepus reduced the browsing time required for locating API usage examples by 46.5%, compared to the time required when using Google. (2) The precision of the search results obtained by Codepus is 91% when using eleven real-world frequently asked questions, which is superior to those of Gists and Open Hub. (3) The average computation time in the search for API usage examples for the eleven queries was only 1.546 seconds with standard deviation of 0.843.

In the future, we will seek to improve the current implementation of Codepus as follows: (1) We will seek to mine a greater number of open source projects for the selection of terms related to code snippets. (2) We will extend the synonym database and optimize the parameters used in the weighting of synonyms. (3) We will include a spell checking feature.

Acknowledgements

This research is sponsored by Ministry of Science and Technology under the grants 103-2221-E-006-218 and 105-2221-E-006-154-MY3 in Taiwan.

References

[1] W. B. Frakes, K. Kang, Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, Vol. 31, No. 7, pp. 529-536, July, 2005.

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

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

[4] S. Haefliger, G. von Krogh, S. Spaeth, Code Reuse in Open Source Software, *Management Science*, Vol. 54, No. 1, pp. 180-193, January, 2008.

[5] 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, S. Wiedenbeck, The State of the Art in End-User Software Engineering, *ACM Computing Surveys*, Vol. 43, No. 3, pp. 21:1-21:44, April, 2011.

[6] O. Hummel, W. Janjic, C. Atkinson, Code Conjurer: Pulling Reusable Software out of Thin Air, *IEEE Software*, Vol. 25, No. 5, pp. 45-52, September-October, 2008.

[7] F. H. Mbuba, W. Y. C. Wang, Software as a Service Adoption: Impact on IT Workers and Functions of IT Department, *Journal of Internet Technology*, Vol. 15 No. 1, pp. 103-114, January, 2014.

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

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

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

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

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

[13] Krugle Search, http://opensearch.krugle.org/

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

[15] Black Duck, Open Hub, https://www.openhub.net/.

[16] W. Maalej, M. P. Robillard, Patterns of Knowledge in API Reference Documentation, *IEEE Transactions on Software Engineering*, Vol. 39, No. 9, pp. 1264-1282, Setpember, 2013.

[17] M. P. Robillard, What Makes APIs Hard to Learn? Answers from Developers, *IEEE Software*, Vol. 26, No. 6, pp. 27-34, November, 2009.

[18] R. Holmes, R. J. Walker, 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, Deccmber, 2006.

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

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

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

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

[23] M. P. Robillard, R. J. Walker, T. Zimmermann, Recommendation Systems for Software Engineering, *IEEE Software*, Vol. 27, No, 4, pp. 80-86, August, 2010.

[24] T. T. Nguyen, H. A. Nguyen, 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*, Amsterdam, Netherlands, 2009, pp. 383-392.

[25] F. Lv, H. Zhang, J Lou, S. Wang, D. Zhang, J. Zhao, CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model, *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, Lincoln, NE, 2015, pp. 260-270.

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

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

[28] S. Thummalapenta, 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*, Chicago, IL, 2007, pp. 204-213.

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

[30] S. Subramanian, L. Inozemtseva, R. Holmes, Live API Documentation, *Proceedings of the International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014, pp. 643-652.

[31] W. Janjic, O. Hummel, C. Atkinson, Reuse-Oriented Code Recommendation Systems, in: M. P. Robillard, W. Maalej, R. J. Walker, Th. Zimmermann (Eds.), *Recommendation Systems in Software Engineering*, Springer, Berlin, 2014, pp. 359-386.

[32] Git Hub Gist, https://gist.github.com/

[33] Code Recommenders, http://eclipse.org/ recommenders/

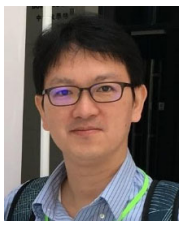
[34] J. Leskovec, A. Rajaraman, J. D. Ullman, *Mining of Massive Datasets*, Cambridge University Press, 2011.

Biographies

Shin-Jie Lee is an associate professor in Computer and Network Center/Department of CSIE at National Cheng Kung University in Taiwan. His 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.

Xavier Lin is a graduate student in Department of Computer Science and Information Engineering at National Cheng Kung University in Taiwan. His research interests include software development and programming.

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



Hsi-Min Chen is an Assistant Professor in the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include software engineering, software architecture, service computing and distributed computing. Chen received his Ph.D. in computer science and information engineering from National Central University, Taiwan.