Enriching Documents with Examples: A Corpus Mining Approach

JINHAN KIM, SANGHOON LEE, and SEUNG-WON HWANG, Pohang University of Science and Technology
SUNGHUN KIM, Hong Kong University of Science and Technology

Software developers increasingly rely on information from the Web, such as documents or code examples on application programming interfaces (APIs), to facilitate their development processes. However, API documents often do not include enough information for developers to fully understand how to use the APIs, and searching for good code examples requires considerable effort.

To address this problem, we propose a novel code example recommendation system that combines the strength of browsing documents and searching for code examples and returns API documents embedded with high-quality code example summaries mined from the Web. Our evaluation results show that our approach provides code examples with high precision and boosts programmer productivity.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Clustering, ranking, code search, API document

ACM Reference Format:
Kim, J., Lee, S., Hwang, S.-W., and Kim, S. 2013. Enriching documents with examples: A corpus mining approach. *ACM Trans. Inf. Syst.* 31, 1, Article 1 (January 2013), 27 pages.
DOI:http://dx.doi.org/10.1145/2414782.2414783

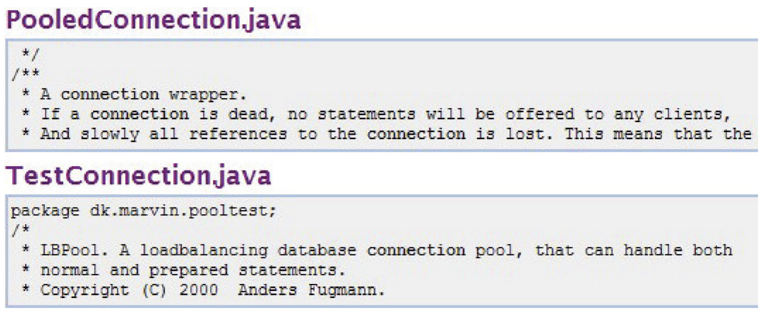
1. INTRODUCTION

As reusing existing application programming interfaces (APIs) significantly improves programmer productivity and software quality [Devanbu et al. 1996; Gaffney and Durek 1989; Lim 1994], many developers search for API information, such as API documents or usage examples, on the Web to understand the usage of APIs.

Specifically, developers may not know which API method to use, in which case they need to browse API documents and read the descriptions before selecting the appropriate one. Alternatively, developers may know which API method to use but may not know how to use it, in which case they need to search for illustrative code examples.

To address these *browsing* needs, API creators usually provide API documents written in a human readable language to show developers how to use the available APIs and select the most suitable one. However, as textual descriptions are often ambiguous and misleading, developers often combine browsing with *searching* for code examples [Bajracharya and Lopes 2009].

This work is based on and extends Kim et al. [2009, 2010].
This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, and the Science and Technology (MEST)/National Research Foundation of Korea (NRF) (Grant 2012-0000474).
Authors' emails: J. Kim, S. Lee, and S.-W. Hwang (corresponding author); {wlsgks08, sanghoon, swhwang}@postech.edu; S. Kim: hunkim@cse.ust.hk.


PooledConnection.java

```
 */
/**
 * A connection wrapper.
 * If a connection is dead, no statements will be offered to any clients,
 * And slowly all references to the connection is lost. This means that the
```

TestConnection.java

```
package dk.marvin.pooltest;
/*
 * LBPool. A loadbalancing database connection pool, that can handle both
 * normal and prepared statements.
 * Copyright (C) 2000  Anders Fugmann.
```

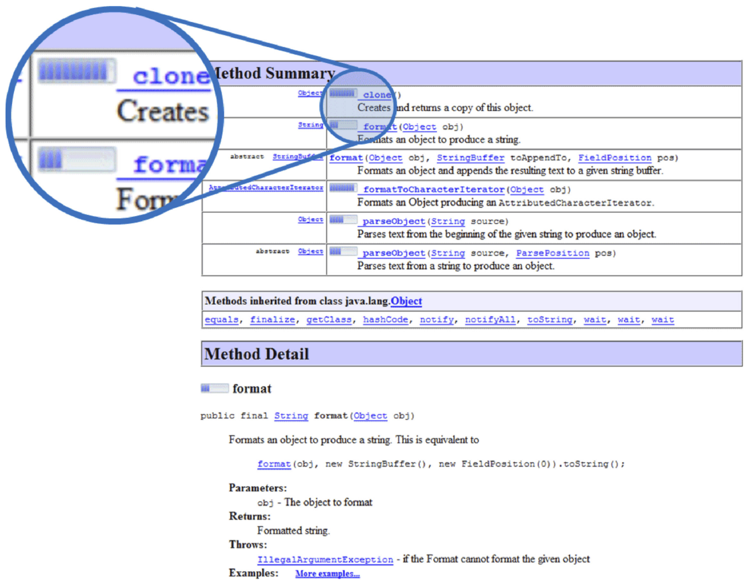
Fig. 1. Top two results from Koders for the query “Connection prepareStatement”.

To address these searching needs, developers often use several of the commercial search engines, including Koders [Koders 2010] and Google Code Search [Google 2010], using an API method name as a query keyword in order to find suitable code examples. However, the top search results from Koders, shown in Figure 1, do not always meet developers’ expectations. The snippets of two top search results show matches in the comments of source code and fail to provide any information about the usage of “prepareStatement().” As a result, human effort is needed to sift through all of the code and find relevant code examples.

To reduce this effort, some API documents, such as the MSDN [MSDN 2010] from Microsoft and the Leopard Reference Library [Leopard 2010] from Apple, include a rich set of code usage examples so that developers do not need to search for additional code examples to understand how to use the API. However, because manually crafting high-quality code examples for all API methods is labor intensive and time consuming, most API documents lack code examples. For example, according to our manual inspection, JDK 5 documents include more than 27,000 methods, but only about 500 of them (approximately 2%) are explained using code examples.

One basic solution to improving API documents is to include code search engine results (code examples) in API documents in advance by leveraging existing code search engines. However, these search engines work well for comment search but fail to retrieve high-quality code examples because they treat code as simple text. Similarly, one may consider code recommendation approaches [Holmes and Murphy 2005; Sahavechaphan and Claypool 2006; Zhong et al. 2009]. However, they require complex contexts, such as the program structure of the current task, in order to recommend suitable code examples.

In clear contrast to these approaches, we propose an intelligent code example recommendation system that searches, summarizes, organizes, and embeds the necessary information in advance by automatically augmenting it with high-quality code examples. Specifically, our system first builds a repository of candidate code examples by interrogating an existing code search engine and then summarizes those candidate code examples into effective snippets. Subsequently, our system extracts semantic features from the summarized code snippets and finds the most representative code examples for which we propose three organization algorithms. Finally, our system embeds the selected code examples into API documents. Figure 2 provides a sample snapshot of generated API documents. Each API method is annotated with popularity, represented by the bar in Figure 2(a), to satisfy developers’ browsing needs to find API methods that are used frequently in programming tasks. To determine the popularity, we count the number of code examples generated using our framework. For each API method, one to five code examples are presented, as shown in Figure 2(b). Hereafter, we will call these documents *example oriented API documents* (*eXoaDocs*).



Method Summary

Object	clone() Creates and returns a copy of this object.
String	format(Object obj) Formats an object to produce a string.
abstract StringBuffer	format(Object obj, StringBuffer toAppendTo, FieldPosition pos) Formats an object and appends the resulting text to a given string buffer.
AttributedCharacterIterator	formatToCharacterIterator(Object obj) Formats an Object producing an AttributedCharacterIterator.
Object	parseObject(String source) Parses text from the beginning of the given string to produce an object.
abstract Object	parseObject(String source, ParsePosition pos) Parses text from a string to produce an object.

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

format

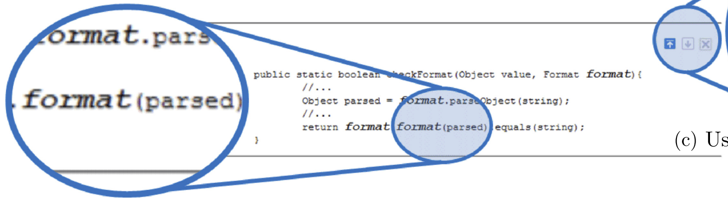
```
public final String format(Object obj)
```

Formats an object to produce a string. This is equivalent to

```
format(obj, new StringBuffer(), new FieldPosition(0)).toString();
```

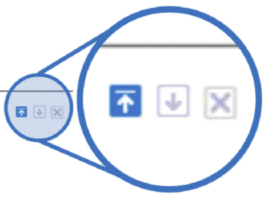
Parameters:
obj - The object to format
Returns:
Formatted string.
Throws:
IllegalArgumentException - if the Format cannot format the given object
Examples: More examples...

(a) Information on the popularity of the methods.



```
public static boolean checkFormat(Object value, Format format){
        //...
        Object parsed = format.parseObject(string);
        //...
        return format.format(parsed).equals(string);
}
```

(b) Code examples.



(c) User feedback button.

Fig. 2. An example page of generated eXoaDocs (Java.text.Format Class).

To evaluate our approach, we first compared our three organization algorithms, which select representative code examples among candidates, and generated eXoaDocs for JDK 5.[1] We then compared our results with the original Java documents (JavaDocs), code search engines, and gold standard results. We further conducted a user study for evaluating how eXoaDocs affect the software development process.

Our evaluation results show that our approach summarizes and ranks code examples with high precision and recall. In addition, our user study indicates that using eXoaDocs boosts programmer productivity and code quality.

The remainder of this article is organized as follows. Section 2 presents our proposed techniques for automatic useful code example generation, Section 3 presents our organization algorithms, and Section 4 evaluates the generated eXoaDocs. A case study and results are presented in Section 5. Section 6 discusses our limitations, Section 7 surveys related work, and Section 8 concludes this article.

[1]In this article, we evaluate our system using JDK 5, but it is easily extensible to other programming languages, such as *C* and *C*$^{++}$, simply by replacing the parser.

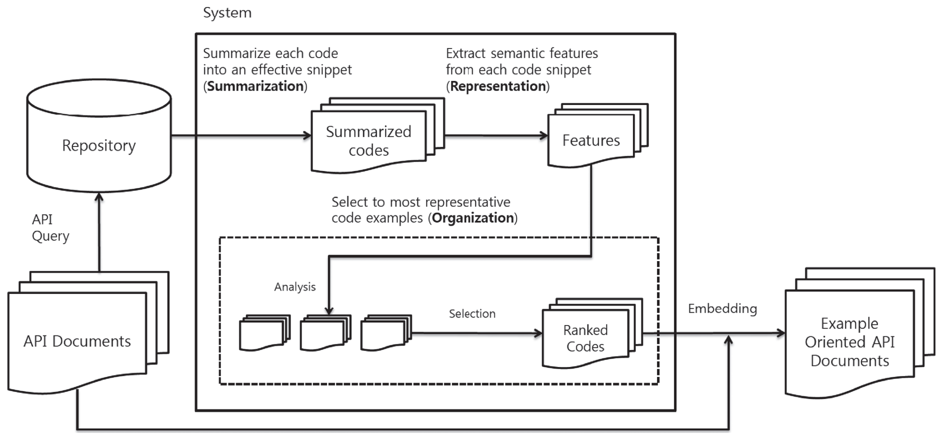

System
Summarize each code into an effective snippet (**Summarization**)
Extract semantic features from each code snippet (**Representation**)
Repository
Summarized codes
Features
Select to most representative code examples (**Organization**)
API Query
Analysis
Selection
Ranked Codes
Embedding
API Documents
Example Oriented API Documents

Fig. 3. Automatic eXoaDocs generation process.

2. OUR APPROACH

In this section, we discuss how to find and provide useful code examples for API documents.

Our framework consists of three modules: Summarization, Representation, and Organization (Figure 3).

First, our framework builds a repository of candidate code examples by leveraging an existing code search engine and summarizing the examples into effective snippets (*Summarization*). After that, it extracts semantic features from each summarized code example (*Representation*). Next, it selects the most representative code examples using clustering and ranking (*Organization*). Finally, it embeds the selected code examples into API documents and generates eXoaDocs.

The following describe the three modules—Summarization, Representation, and Organization—in detail.

2.1. Summarization

The first module of our framework collects potential code examples for each API method and builds a code example repository.

To achieve this, we first leverage a code search engine, Koders, by querying the engine with the given API method name and its interface name extracted from API documents. For example, we query 'Connection PrepareStatement' to collect source code of the API method *PrepareStatement* in the interface *Connection*. We then collect the top 200 source code files for each API method. We selected 200 as the retrieval size based on our observation that most results ranked lower than the top 200 are either irrelevant to the query or redundant. We will further discuss the pros and cons of leveraging code search engines for populating the repository in Section 6. After collecting the top 200 source code files, we ignore the Koders ranking and only use these files to construct a repository. We then use our own summarization and ranking algorithms to identify suitable code examples.

Next, we summarize the code as *good example snippets* that clearly explain the usage of the given API method. To obtain initial intuitions on defining good example snippets, we first observed manually designed code examples in JavaDocs. We illustrate the observed characteristics of good example snippets using a manually written code example to explain the *add* method in the *Frame* class, as shown in Figure 4.

```
1  public static void main(String [ ] args){
2       Frame f = new Frame("GridBagLayout Ex.");
3       GridBagEx1 ex1 = new GridBagEx1();
4       ex1.init();
5       //...
6       f.add("Center", ex1);
7  }
```

Fig. 4. A manually constructed code example in JavaDocs.

— Good example snippets should include the corresponding API method call, for example, *f.add()* in line 6 of Figure 4, and its semantic context, such as the declaration and initialization of an argument *ex1* (lines 3 and 4) within the method.
— Irrelevant code, regardless of the textual proximity to the API method, can be omitted, as in line 5 in the manually written example.

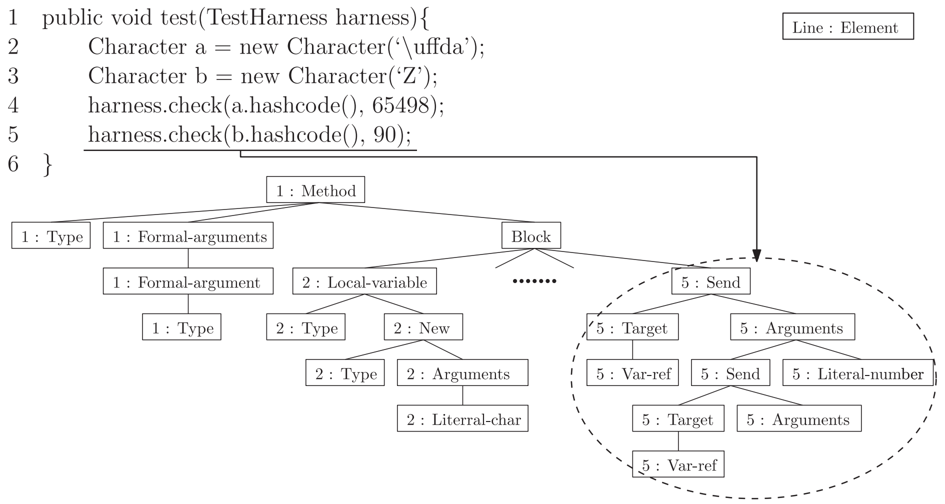
Recall that both of the top two code snippets from Koders in Figure 1 violate all of the characteristics needed to be good example snippets by (1) failing to show the actual API method call and (2) summarizing simply on the basis of the proximity of text to query keyword matches, *Connection* and *PrepareStatement*.

In clear contrast to the code search engine results, we achieve summarization based on the semantic context by following these steps.

— *Method extraction.* We identify the methods that contain the given API method call because they show the usage and semantic context of the given API method call.
— *API slicing*. We extract only the semantically relevant lines for the given API method call using slicing techniques [Horwitz et al. 1988; Weiser 1981]. Relevant lines need to satisfy at least one of the following requirements: (R1) declaring the input arguments for the given API method, (R2) changing or initializing the values of the input arguments for the given API method, (R3) declaring the class of the given API method, or (R4) calling the given API method. For example, in Figure 4, line 2 is relevant because it declares the *Frame* class of the API method (R3); line 3, because it declares the input argument *ex1* (R1); line 4, because it initializes the input argument (R2); and line 6, because it calls the given API method (R4). All other lines were omitted as being irrelevant.

To identify these relevant lines, we first analyze the semantic context of the identified methods by building an abstract syntax tree (AST) from the potential code example using an open source tool, java2xml [Java2Xml 2010]. To illustrate, Figure 5 shows a code example and its AST with the line information. Each node of the AST represents an element of the Java language. For example, line 5 of the code example is converted to the subtree in the dotted circle. In that subtree, the node *Send* is a Java element indicating the actual API method call— the first *Send* at the root of the subtree represents *check*, and the second represents *hashcode*. *Target* and *Var-ref* indicate the interface and its name for the API method. For instance, the left descendant of the root, *Target*, corresponds to the interface of the *check* API method call, which is *harness*. Its right descendant, *Arguments*, has two descendants representing two arguments, *b.hashcode()* and *90*.

Next, to effectively find the relevant lines from the AST, we design a simple lookup table, called the Intra-Method Analysis Table (IAT). To build the IAT, we traverse the AST and examine each element in the node. If the element in the node indicates variables or API method calls, we store the name and type of the element and the line number on which the element is called. When we find elements related to API



```
1  public void test(TestHarness harness){
2       Character a = new Character('\uffda');
3       Character b = new Character('Z');
4       harness.check(a.hashcode(), 65498);
5       harness.check(b.hashcode(), 90);
6  }
```

Fig. 5. eXoaDocs code example of "Character.hashcode()" and its abstract syntax tree.

Table I. An Example of the Intra-Method Analysis Table (IAT) for the Code in Figure 5

Line	Interface	API name	Arguments
4	a : Character : 2	hashCode	
5	b : Character : 3	hashCode	

method calls while traversing nodes in the AST, we check the stored elements, obtain the elements related to the API method call (e.g., variables used in the API method call), and store them in the IAT.

Each row in an IAT corresponds to an API method call and consists of four parts, as shown in Table I: the line number of the API method, the name and line number of the interface of the API method, the API method name, and the type and line information of the argument list of the API method. For example, the first row of Table I shows that (a) "hashCode()" is called at line 4, (b) its interface type is "Character" with name "a" (declared at line 2), and (c) "hashCode()" has no argument.

As an IAT contains all Java language elements related to the API method call, our system can find all relevant lines of the API method call by simply searching the rows of the IAT and checking API method calls. Our system then summarizes the original source code and generates example snippets using only semantically relevant lines in the IAT.

2.2. Representation

The goal of representation is to extract meaningful features from each summarized code example to identify the most representative code examples by comparing their features.

There are two extreme approaches to extracting features from each code example and comparing extracted features between code examples. One extreme is to treat code examples as simple texts, while the other extreme is to represent them as ASTs. As the former compares textual features between code examples based on keyword matching, it is highly efficient but neglects the semantic context. Conversely, as the latter constructs the ASTs using the semantic context of code examples and compares the AST between code examples, it is very expensive but considers the semantic context.

We find an intermediate approach between these two extremes by approximating the semantic context and comparing the approximated semantic context (a trade-off between efficiency and accuracy). To approximate the semantic context, we extract semantic context vectors from the ASTs using a clone detection algorithm called DECKARD [Jiang et al. 2007].

DECKARD proposes *q-level characteristic vectors* for approximating the semantic context of AST for each code, where q is the depth of the AST subtree. When the value of q is fixed, DECKARD extracts all possible AST subtrees with depth q and generates q-level characteristic vectors consisting of an n-dimensional numeric vector $< c_1, ..., c_n >$, where n is the number of elements in the subtree, such as *loops*, and each c_i is the number of occurrences of a specific element. DECKARD selects some relevant elements in order to generate q-level characteristic vectors. For example, when computing the vector for the subtree with line 5 of the code in Figure 5, that is, the tree segment in the dotted circle, the characteristic vector for selected elements such as *send*, *arguments*, *local-variable*, and *new*, would be $< 2, 2, 0, 0 >$, because the subtree contains two *send* and two *arguments* but does not contain *local-variable* and *new*.

Similarly, we use all 85 Java language source code elements, detected from ASTs using "java2xml." (For the complete listing of all 85 elements, refer to `http://exoa.postech.ac.kr`.) The AST in Figure 5 contains a part of the 85 Java language source code elements we used, such as *method*, *formal-argument*, *local-variable*, *arguments*, and *send*. These 85 features have integer values that represent the count for each element in the given code example.

In addition, we consider two more integer features specific to our problem context of code example recommendation. First, we add the *frequency* of the query API method in each code example, as there is a higher chance that a code example that has many query API method calls will be an example that the user is searching. For this reason, we count the number of query API method calls in the code example and use it as a feature. Second, we add the the number of lines of the code examples as a feature to filter lengthy code examples, as developers usually prefer concise code examples. In this article, the length of the code example is the length of the entire method for code examples generated by "Method extraction" and the length of the code snippet for code examples generated by "API slicing."

In total, we use 87-dimensional feature vectors to represent the code examples.

2.3. Organization

In this section, we discuss how to organize code examples to best satisfy user intent.

To satisfy user intent, we need to know what users want. However, user intent is often unclear, as users may have different usage types in mind, and it is impossible to guess the usage type using only the API method name. In a similar context, general search engines have taken one of the following two approaches when the query intent is ambiguous.

— *Clustering*. Clusty[2] and Carrot[3] cluster search results into multiple groups satisfying diverse user intents and present the representatives of each group.
— *Ranking*. Typical search engines estimate the probability of user intent and rank to optimize for the most likely intent.

These two approaches have complementary strengths, as summarized in Table II. As the clustering-based approach finds clusters satisfying diverse user intents, code examples presented can cover a diverse range of usage types. However, this approach

[2]http://clusty.com/
[3]http://search.carrot2.org/stable/search

Table II. Complementary Strength of Ranking and Clustering-Based Approaches

Algorithm	Pros	Cons
Clustering-based approach	cover diverse usage types	results may include outlier types
Ranking-based approach	consider different probabilities of intents	results may be skewed

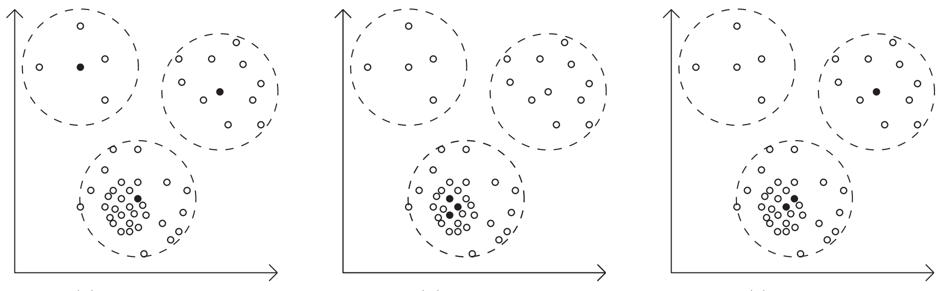

(a) eXoaCluster. (b) eXoaRank. (c) eXoaHybrid.

Fig. 6. Illustrations of presentation algorithms.

may return outliers by not considering probabilities of user intention, thus and generating a group for an intent that is extremely unlikely. Conversely, the ranking-based approach considers such probabilities and optimizes for the most likely user intent, which reduces the coverage. As a result, it provides the most widely used code examples. However, the results tend to skew to satisfy a few popular intents.

We examined both ranking- and clustering-based approaches for the organization of code examples and propose the following three algorithms. To illustrate each algorithm, Figure 6 shows an example that plots candidate code examples in two-dimensional space (even though each code example is 87-dimensional), where the dotted circles are clusters representing the usage types of an API method and three black dots are code examples representing the code examples chosen by each algorithm. Note that the distance between points reflects the code similarity between two code examples.

— `eXoaCluster` (Figure 6(a)) adopts a clustering-based approach. It identifies clusters of code examples representing the same usage type and selects the most representative code example from each cluster. As a result, `eXoaCluster` provides code examples with diverse usage types.
— `eXoaRank` (Figure 6(b)) adopts a ranking-based approach. It first estimates the probability of each intent from the code corpus, assuming, that the most widely used usage type is likely to be asked. Specifically, the probability of each intent is estimated using centrality [Erkan and Radev 2004], representing highly likely intents, inspired by document summarization literature. `eXoaRank` then selects the top-k representative code examples with high centrality.
— `eXoaHybrid` (Figure 6(c)) balances the previous two algorithms using their complementary strengths by not only considering the probability of each intent but also trying to diversify the results over multiple usage types. `eXoaHybrid` iteratively computes the marginal utility [Agrawal et al. 2009] by using the probabilities of clusters and code examples to satisfy the user intents. It then selects the code example with the highest marginal utility at each iteration. `eXoaHybrid` thus recognizes that the cluster size and the number of representatives from each cluster can be different.

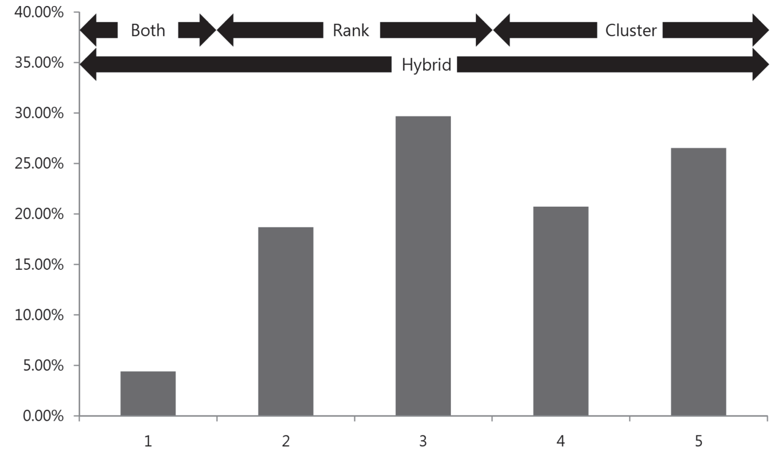


Fig. 7. Distribution of the number of different usage types in API methods in JDK 5. For highly diverse API methods (4∼5), "Cluster" provides better code examples, while for less diverse API methods (2∼3), "Rank" provides better code examples. As "Rank" and "Cluster" are identical when API usage is not diverse (or the number of usage types is 1), we mark this as "both."

Figure 7 shows the distribution of the number of usage types found in API methods in JDK 5. Observe that such a number is in the range from one to five, with five suggesting that the API method has five different usage types (i.e., diverse) and, in another extreme, the API method has a single usage type every one agrees with (i.e., not diverse). Intuitively, for highly diverse API methods, clustering the results by types and presenting different usages would be more reasonable. Meanwhile, in another extreme, every result falls into a single cluster such that clustering becomes a redundant phase. Our experimental results are consistent with this intuition. Figure 7 shows that for diverse scenarios with API methods with 4+ usage types, `eXoaCluster` is the winner, and for the less diverse scenarios, `eXoaRank` is the winner. Approximately half of API methods fall into the former and the rest into the latter scenario.

So, of what use is `eXoaHybrid`? If it is easy to predict whether the given API method is diverse, selecting the winning algorithm for either case is the most effective. However, in a case where such a prediction is non-trivial, `eXoaHybrid` closely approximates the accuracy of the winner in all scenarios, while the accuracy of `eXoaRank` deteriorates in diverse scenarios (and so does that of `eXoaCluster` in less diverse scenarios). That is, `eXoaHybrid`, which balances `eXoaCluster` and `eXoaRank`, can be the best choice to satisfy developers' intent for diverse API methods, as discussed in detail in Section 5.

The technical details of each algorithm will be discussed in the next section.

3. ORGANIZATION ALGORITHMS

In this section, we describe the technical details of our three proposed algorithms–`eXoaCluster`, `eXoaRank`, and `eXoaHybrid`.

3.1. eXoaCluster

The goal of `eXoaCluster` is to cluster code examples into groups of similar usage types and select the representatives from each group.

3.1.1. Clustering. The first step of `eXoaCluster` is to cluster the code examples into groups of different usage types, such that our results can cover various usage types of the given API method. Because we represent code examples as characteristic vectors, as described in Section 2.2, well known clustering algorithms on numerical data, such as the k-means algorithm, can be applied using *L1*-distance as the similarity metric defined as the following.

$$L1\text{-distance}(V_1, V_2) = \sum_{i=1}^{n} |x_i - y_i|,$$

where V_1 and V_2 are n-dimensional characteristic vectors and x_i and y_i are the ith elements of V_1 and V_2, respectively. In our problem, k is both the number of clusters and the number of usage types of the given API method.

However, the key challenge is that the number of clusters k varies over different API methods, which requires the automatic tuning of k during a query. However, applying existing algorithms that support this tuning, such as X-means [Pelleg and Moore 2000], renders poor clustering results for our problem with 87-dimensional data. The problem is that these algorithms iteratively look for the best split and check whether the result of the split improves cluster quality. However, as all of the splits become near-equally desirable in the high-dimensional space (known as the “curse of dimensionality” problem [Xu and Wunsch 2005]), X-means assigns all code examples to one cluster or one code example to one cluster. In other words, the number of clusters tuned by X-means either remains k_{min} or exponentially increases and terminates when $k = k_{max}$ in all cases, where k_{min} is the minimum number of clusters and k_{max} is the maximum number of clusters (default values are $k_{min} = 1$ and k_{max} = number of code examples).

In clear contrast, in our problem context, the number of usage types is usually small. We empirically observed that the number of usage types is usually in the range from 2 to 5. Using this observation, we invoke the k-means clustering algorithm four times, that is, for $k = 2, 3, 4$, and 5, and select the results with the best clustering quality. While this approach is generally discouraged for incremental splitting as used in X-means, we observe that it achieves higher efficiency and accuracy compared to X-means in our specific problem setting with (1) a tightly bounded k range and (2) high-dimensional data.

To select the best clustering results, we use three cluster quality metrics, as similarly defined in X-means. We compute the following scores for each clustering result, sum those scores, and select the clustering result with the highest score. To avoid underrepresented clusters, we exclude clusters with just one or two code examples.

— *Centroid distribution*. Clustering where centroids are distributed evenly suggests a good coverage of varying usage types, as the centroid of a cluster represents the characteristic of the cluster. We check whether centroids of clusters are evenly distributed. We measure $\frac{1}{var_i}$ for variance var_i of centroids from the clustering results, where i is the number of clusters.
— *Sum of the squared error (SSE)*. If code examples are categorized well, then they should have similar characteristic vectors with the centroids of their clusters. We check SSE based on the distance between each code example and the centroid of its cluster. As SSE usually decreases as the number of clusters increases, we compute the decreasing quantity. A large decrease in SSE when the number of clusters is increased suggests that centroids are statistically better representatives of all vectors in the cluster, indicating high-quality clustering results. We measure this as $\Delta_i = \text{SSE}_{i-1} - \text{SSE}_i$, which represents the difference in SSE, where $i-1$ and i are the number of clusters.
— *Hierarchical clustering*. The clusters that represent usages typically exhibit a hierarchical structure. In other words, a cluster can be split into subtypes (the lower layers in the hierarchy) or two clusters can be merged to represent their supertype

(the higher layer). To favor clustering results that preserve this hierarchical relationship, we give a score of 1 if all members of each cluster (when the number of clusters is i) come from one cluster in the results when the number of clusters is $i-1$, and 0 otherwise.

We choose the k value that yields the highest score, which is the sum of scores, and k determines the number of usage types to present for each API method.

3.1.2. Code Example Selection. Once the clusters are identified, we compute the following three normalized scores in the range [0, 1] as there is no weight factor between scores. Next, we add those scores, rank code examples using aggregated overall scores, and select the most representative code example from each cluster with the highest overall score.

— *Representativeness*. Representative code examples should have similar characteristic vectors to their cluster's centroid. This representativeness can be computed by using *L1*-distance [Campbell 1984; Foody et al. 1992]. Specifically, the measure of the representativeness is $\min(\frac{1}{similarity}, 1)$.
— *Conciseness*. Concise code examples have better readability than lengthy ones, and so developers usually prefer concise code examples. For this reason, we give higher scores to concise code examples, that is, those with fewer lines of code. Specifically, the measure of the conciseness is $\frac{1}{\mathrm{length}}$.
— *Correctness*. Suitable code examples should contain the correct type of classes and arguments. However, API methods with the same name can belong to different classes or can take different arguments. In addition, IAT may miss the correct type of classes or arguments and return non-type information. If there are code examples that contain the correct type information, we do not provide code examples with non-type information. However, if there is no such correct code example, code examples with non-type information should be presented. For this reason, we give a high score to code examples that use API methods that have the correct classes and the matching of arguments. Specifically, we give the score 1 when the code example has correct information, and 0 otherwise.

3.2. eXoaRank

As an alternative approach to `eXoaCluster` that shows the representatives of all clusters, treating all clusters as equally to be chosen by developers, we design `eXoaRank` by considering the different probabilities that code examples will satisfy the developers.

We estimate the probability of each intent from the code corpus, assuming that the usage type most commonly used by developers is most likely to be asked in the query. Specifically, we compute such probability as *centrality* [Erkan and Radev 2004], measuring how many similar code examples are contained in the corpus. Therefore, a high centrality score for an API method means that there are many similar code examples used by developers. In other words, developers commonly use the API in the way presented in the specific algorithm. That is, this code example is likely to satisfy the user intent with high probability.

To measure the centrality, we model all code examples in the code corpus as a graph, where each node represents a code example and is connected by edges with weights representing pairwise similarity. We quantify the similarity using the $L1$-distance between characteristic vectors, as described in Section 2.2.

$$L1(v_i, v_j) = \sum_{d=1}^{n} |v_i(d) - v_j(d)|,$$

where v_i and v_j are the characteristic vectors of two code examples, n is the dimensionality of the vectors, and $v_i(d)$ is the dth value of v_i. Using the $L1$-distance, we compute the normalized similarity between two code examples.

$$\text{Similarity}(v_i, v_j) = 1/L1(v_i, v_j).$$

This similarity is between 0 and 1. It is higher for similar pairs. For identical pairs, because we cannot compute the similarity score as it approaches infinity, we give the maximum score of 1. To keep the graph structure compact, we only retain the edges that have a weight higher than a certain threshold.

Once the graph is constructed, we compute the centrality score of each node (or code example). A naive approach would be simply to count the number of edges connected.

$$\text{Centrality}_0(c_i) = \text{the number of edges connected with } c_i,$$

where c_i is a node. However, centrality has a recursive nature, and hence nodes that are similar to those with high centrality will also have high centrality, as similarly reflected in metrics in social networks, such as Web graphs or co-authorship graphs (e.g., PageRank [Page et al. 1999]). To reflect this, we use the preceding metric as initial values at iteration 0 and iteratively refine the centrality value, $\text{Centrality}_t(c_i)$, at each iteration t, as shown in the following.

$$\text{Centrality}_t(c_i) = \sum_{c_j \in S(c_i)} \text{Centrality}_{t-1}(c_j),$$

where $S(c_i)$ is the set of code examples that are connected with c_i. To avoid outliers, we extend the centrality notion to be divided by the number of edges.

$$\text{Centrality}_t(c_i) = \sum_{c_j \in S(c_i)} \frac{\text{Centrality}_{t-1}(c_j)}{\text{\# of edges of } c_j}.$$

We compute the centrality scores of nodes iteratively until they converge and present the top-k results with the highest centrality scores.

3.3. eXoaHybrid

As illustrated in Figure 6, cluster- and ranking-based approaches have complementary strengths: a cluster-based approach maximizes the coverage but is agnostic to probabilities, while a ranking-based approach is probability aware but provides results that may skew to a few popular types.

In this section, we describe `eXoaHybrid`, which balances the strengths by considering the probability of centrality but tries to diversify the results over multiple usage types by avoiding redundancy, as similarly studied in Agrawal et al. [2009] for diversifying general search results.

We use the top-k representatives selected from `eXoaRank` and `eXoaCluster`. Given $2k$ selected code examples, we first assign them to k clusters generated by `eXoaCluster`.

We then iteratively select a code example for presentation until k code examples have been selected. To select representative code examples, we first quantify the probability of each cluster satisfying the user intent. The probability of each cluster, denoted as $P(C_i|API)$, is defined as follows.

$$P(C_i|API) = \frac{\text{Number of selected code examples in } C_i}{\text{Total number of selected code examples}},$$

where C_i is the ith cluster and API is a given specific method.

Table III. An Example Dataset

Code	Cluster	$P(C_i\|API)$	$V(e\|API, C_i)$
e_1	C_1	0.2	0.9
e_2	C_2	0.2	0.2
e_3	C_3	0.6	0.8
e_4	C_3	0.6	0.6
e_5	C_3	0.6	0.5

Table IV. Illustration of `eXoaHybrid` Algorithm Using the Dataset in Table III

Iteration	Code examples	Marginal utilities	Result set
1	$\{e_1, e_2, \mathbf{e_3}, e_4, e_5\}$	$\{0.18, 0.04, \mathbf{0.48}, 0.36, 0.30\}$	$\{e_3\}$
2	$\{\mathbf{e_1}, e_2, e_4, e_5\}$	$\{\mathbf{0.18}, 0.04, 0.072, 0.06\}$	$\{e_3, e_1\}$
3	$\{e_2, \mathbf{e_4}, e_5\}$	$\{0.04, \mathbf{0.072}, 0.06\}$	$\{e_3, e_1, e_4\}$

After computing the probability of the cluster, which represents the frequency of the usage types, we compute the probability of each code example in each cluster to select suitable code examples in the cluster. Let $V(e|API, C_i)$ be the probability that the code example e in cluster C_i for the given API satisfies developers. In our article, we use the following equation.

$$V(e|API, C_i) = \alpha \times RS_C + \beta \times RS_R,$$

where RS_C is the normalized `eXoaCluster` rank score (Section 3.1.2) of e; RS_R is the normalized `eXoaRank` rank score (Section 3.2) of e; and α and β are weight parameters, as both `eXoaCluster` and `eXoaRank` scores represent the quality of the code example e. Both scores are normalized in the range $[0, 1]$.

Using $P(C_i|API)$ and $V(e|API, C_i)$, we compute the overall probability of the code examples. As $P(C_i|API)$ represents the probability of the cluster C_i satisfying the user intent and $V(e|API, C_i)$ indicates the probability that the code example e in C_i satisfies the user, the total probability of code example e can be computed as follows.

$$P(C_i|API) \times V(e|API, C_i).$$

We select the code example with the highest probability value.

However, ranking examples by the preceding value will retrieve a redundant set of highly similar results, as in `eXoaRank`. For diversification, conditional probability was defined in Agrawal et al. [2009], with respect to the set S of code examples already selected by `eXoaHybrid` in C_i. Initially, when none has been selected, that is, $S = \emptyset$, the conditional probability U is identical to the probability of cluster P, that is, $U(C_i|API, \emptyset) = P(C_i|API)$. However, when some code example e from C_i is selected, $U(C_i|API, S)$ is updated to

$$U(C_i|API, S \cup \{e\}) = U(C_i|API, S) \times (1 - V(e|API, C_i)).$$

With U, we rank the code examples by the *marginal utility* in order to obtain diversified results.

$$g(e|API, C_i, S) = U(C_i|API, S) \times V(e|API, C_i).$$

To illustrate how `eXoaHybrid` works for our example ranking, Table IV shows an example using the dataset in Table III. In this example, the initial marginal utilities of the code examples are 0.18, 0.04, 0.48, 0.36, and 0.30. Because e_3 has the highest marginal utility, `eXoaHybrid` selects e_3. Once this code example is selected, the marginal utilities of the other code examples in C_3 decrease, because conditional probability $U(C_3|API, \{e_3\})$ is updated to $0.12 = 0.6 \times (1 - 0.8)$. After this update, the

Table V. Agreement of Selected Code Examples in the First Gold Standard Set

	assessor 1	assessor 2	assessor 3
assessor 1	1.0000	0.5814	0.6452
assessor 2	—	1.0000	0.5224
assessor 3	—	—	1.0000

Note: The average of agreements is 0.5830.

marginal utilities of $\{e_1, e_2, e_4, e_5\}$ are $\{0.18, 0.04, 0.072, 0.06\}$. At the second iteration, `eXoaHybrid` selects e_1 with the highest marginal utility, and $U(C_1|API, \{e_1\})$ decreases to $0.02 = 0.2 \times (1 - 0.9)$. At the third iteration, among the examples $\{e_2, e_4, e_5\}$ of marginal utilities $\{0.04, 0.072, 0.06\}$, `eXoaHybrid` selects e_4. Finally, `eXoaHybrid` returns $\{e_1, e_3, e_4\}$ as the top three code examples. In contrast, as `eXoaCluster` selects representative code example from each cluster, it will return $\{e_1, e_2, e_3\}$. Furthermore, as `eXoaRank` selects code examples with the highest centrality and cluster C_3 has three code examples that have similar usage, it will return $\{e_3, e_4, e_5\}$.

4. EVALUATION

In this section, we first evaluate the quality of the three proposed organization algorithms using generated eXoaDocs for JDK 5. We then evaluate the quality of our search results by comparing them with existing API documents, code search engines, and gold standard results. The eXoaDocs used for this evaluation are provided at `http://exoa.postech.ac.kr`.

4.1. Comparison of Organization Algorithms

In this section, we compare the quality of code examples selected by the three proposed algorithms: `eXoaCluster`, `eXoaRank`, and `eXoaHybrid`.

For comparison purposes, we constructed two gold standard sets. We first randomly selected 50 API methods. For each API method, the three proposed algorithms selected the top-k representatives, where k is the number of usage types (or clusters) found by `eXoaCluster`. We then divided them into two sets. The first gold standard set consisted of API methods with four or more usage types (or large number of clusters), and the second gold standard set consisted of API methods with two or three usage types (or small number of clusters). As a result, 22 API methods were in the first set and 20 API methods were in the second set. After that, we collected the top-k representatives from `eXoaRank` and `eXoaCluster`, respectively, from each API method. As some code examples were selected by both `eXoaRank` and `eXoaCluster`, we collected between k and $2k$ representatives from each API method. Naturally, these selected results contained the top-k results of `eXoaHybrid` (we set $\alpha = 0.2$ and $\beta = 0.8$ based on empirical observation). As a result, 198 candidate code examples were selected from the first set and 117 from the second set. These candidate code examples were then presented to three human assessors who were asked to pick the best k results from each API method.

Tables V and VI present the agreement matrix among the three human assessors. The agreement was quantified using the Jaccard similarity [Jaccard 1901], such that the agreement between two result sets R_A and R_B is defined as follows.

$$\text{Agreement}(R_A, R_B) = \frac{|R_A \cap R_B|}{|R_A \cup R_B|}.$$

This metric scores higher for two sets with higher agreements, for example, 1 when $R_A = R_B$ and 0 when disjoint. Observe from Tables V and VI that all agreements between any two assessors are greater than 0.5, and the average agreements are 0.5830 and 0.6854.

Table VI. Agreement of Selected Code Examples in the Second Gold Standard Set

	assessor 1	assessor 2	assessor 3
assessor 1	1.0000	0.7500	0.6060
assessor 2	—	1.0000	0.7000
assessor 3	—	—	1.0000

Note: The average of agreements is 0.6854.

To validate the reliability of the agreements, we computed the Fleiss' kappa score [Fleiss 2010], which is a statistical measure of the reliability of agreement among a fixed number of assessors. Let N be the number of code examples, n be the number of assessors, and k be the number of categories (in our case for the first gold standard set, $N = 198$, $n = 3$, and the category is *Selected* or *Not Selected*). To compute Fleiss' kappa κ, we first computed p_j, the proportion of all assignments that were made to the jth category.

$$p_j = \frac{1}{Nn}\sum_{i=1}^{N} n_{ij},$$

where n_{ij} is the number of assessors who assigned the ith code example to the jth category. We next computed P_i, which indicates the extent of agreement among assessors for the ith code example.

$$P_i = \frac{1}{n(n-1)}\sum_{j=1}^{k} n_{ij}(n_{ij}-1).$$

Using p_j and P_i, κ is defined as

$$\kappa = \frac{\bar{P}-\bar{P_e}}{1-\bar{P_e}},$$

where $\bar{P}$ is the mean of P_i, and $\bar{P_e}$ is the sum of the square of p_j. The factor $1-\bar{P_e}$ indicates the maximum agreement score, and $\bar{P}-\bar{P_e}$ indicates the achieved agreement score. This metric also scores higher with higher agreement, for example, 1 when all assessors completely agree with each other, but equal to or lower than 0 when disjoint. Fleiss' kappa scores of the two gold standard sets are 0.4540 and 0.4864, respectively. These scores are interpreted as having moderate agreement [Fleiss 2010], which indicates that agreements among the three assessors are reliable.

On the basis of these assessments, we built the two gold standard answer sets using majority voting, by selecting the candidates picked by two or more assessors. As a result, from the first gold standard set, which consisted of four or more usage types, 107 code examples were selected as the gold standard answers among 198 candidate code examples, and from the second gold standard set, which consists of two or three usage types, 82 code examples were selected as the gold standard answers among 117 candidate code examples. On the basis of these gold standard sets, denoted as R_{G1} and R_{G2}, we computed the agreement between each gold standard set and the result set from each algorithm R_A, that is, $Agreement(R_G, R_A)$, as shown in Tables VII and VIII.

`eXoaCluster` showed the best performance among the three proposed algorithms (0.6328 agreement) for JDK 5 API methods that had four or more usage types, while `eXoaRank` showed the best performance (0.4615 agreement) for JDK 5 API methods that had two or three usage types.

Table VII. Agreement Score between the First Gold Standard Set and Each Organization Algorithm

	`eXoaCluster`	`eXoaRank`	`eXoaHybrid`
R_{G1}	**0.6328**	0.1808	0.5145

Table VIII. Agreement Score between the Second Gold Standard Set and Each Organization Algorithm

	`eXoaCluster`	`eXoaRank`	`eXoaHybrid`
R_{G2}	0.3168	**0.4615**	0.3434

We first analyzed the reason why `eXoaCluster` showed the best performance for R_{G1} and `eXoaRank` showed the best performance for R_{G2} by analyzing the three human assessors' selections.

The results are consistent with our intuition, discussed in Section 2.3. For API methods with few usage types, the usage types selected by the human assessors are also skewed to one homogeneous type, which well explains why `eXoaRank` is the winner. For API methods with diverse usage types, the human assessors' decisions varied over types, for which `eXoaCluster`—showing diverse clusters—is effective. Meanwhile, `eXoaHybrid` closely emulates the performance of the winning algorithm for all API methods. Considering that it is hard to predict how diverse the use case is per API method, `eXoaHybrid` can be a practical choice for generating examples for all types. More specifically, in the evaluation using R_{G1}, in many cases, `eXoaHybrid` effectively selected the gold standard results that were not selected by `eXoaCluster` (11 API methods among 22 API methods). `eXoaHybrid` also selected the same number of gold standard results as `eXoaCluster` for 10 API methods. In the evaluation using R_{G2}, `eXoaHybrid` effectively selected the gold standard results that were not selected by `eXoaRank` (13 API methods among 20 API methods) and also selected the same number of gold standard results as `eXoaRank` for 12 API methods. These results indicate that `eXoaHybrid` effectively filtered out outliers from `eXoaCluster` and selected more representative code examples from `eXoaRank`.

For the remaining evaluations, we used `eXoaCluster`, which showed the highest average agreement score between two gold standard test results (Tables VII and VIII).

4.2. Comparison with JavaDocs

We compared the number of code examples that `eXoaDocs` generated for JDK 5 with that generated by JavaDocs.

To determine the number of code examples generated by JavaDocs, we analyzed an HTML source code. As the $< pre >$ tag of HTML contains code information, we checked each method to ascertain whether it contained a $< pre >$ tag. However, as other contents are also expressed using a $< pre >$ tag, we manually checked whether the the contents of the $< pre >$ tag were code examples.

Out of more than 27,000 methods, `eXoaDocs` augmented 75% of code examples (more than 20,000 methods), while only 2% (about 500 methods) were augmented in JavaDocs.

This result shows that our system provides a rich set of code examples when the API document contains only a small number of code examples. In addition, even if the API document already contains a rich set of code examples, our system can provide additional real-life code examples using the APIs.

4.3. Comparison with Code Search Engines

In this section, we compare the quality of `eXoaDocs` with two code search engines, namely Koders and Google Code Search.

Table IX. Top Query Results for Koders, Google Code Search, and eXoaDocs Using 10 Randomly Selected API Methods

	Relevant code and snippet	Relevant code	Irrelevant code
Koders	22%	8%	70%
Google	12%	10%	78%
eXoaDocs	92%	6%	2%

Note: Most of the Koders and Google Code Search results were irrelevant, but most of the eXoaDocs results were relevant and had a good snippet.

We queried with ten randomly selected API methods and manually divided the top-k results (where k is the number of examples embedded in eXoaDocs) of Koders, Google Code Search, and eXoaDocs into the following three groups, based on the criteria described in the following.

— *Relevant code and snippet.* The code includes the correct API method usage, and its snippet correctly summarizes the API method usage. As a result, the appropriate API method and all parameters used in the API method are well explained in the snippet (i.e., a good ranking and good summarization).
— *Relevant code.* The code includes the correct API method usage, but its snippet incorrectly summarizes the API method usage. As a result, the appropriate API method or some parameters used in the API method are not explained in the snippet (i.e., a good ranking but bad summarization).
— *Irrelevant code.* In this case, the queried API method does not appear in either the code or its summarized snippet (i.e., a bad ranking and bad summarization.

Table IX presents our evaluation results. As eXoaDocs find top results by analyzing the semantic context of source code, most of the eXoaDocs results (92%) showed proper API method usage in the summary, while only 22% and 12% of the snippets in Koders and Google Code Search satisfied the requirements. In addition, the vast majority of the results from Koders and Google Code Search, 70% and 78% respectively, were irrelevant (showing comment lines or import statements, which are not helpful). This lack of relevance is explained by the fact that when textual features are being built, they cannot distinguish between relevant and irrelevant matches.

4.4. Comparison with Gold Standard

We also evaluated the quality of the code examples of eXoaDocs in terms of ranking and summarization.

Ranking precision/recall. For ranking precision and recall, we first constructed a ranking gold standard. To construct the ranking gold standard, we randomly selected 20 API methods. As each API method contained about 80 code example summaries on average and it was hard to evaluate all of them, we collected $2k$ code examples from each API method, where k is the number of usages (or clusters) of each API method. The $2k$ code examples consisted of the top-k summaries selected by eXoaDocs and other k summaries not selected by eXoaDocs among all candidate summaries. As a result, 130 code examples were selected from 20 API methods, and these code examples were presented to five human assessors. We asked each human assessor to mark the k best code examples from each API and determined the gold standard set using *majority voting*, marked as R_G. As a result, 41 code examples were selected as R_G. To measure

```
01 public class A{
02     public static void main(){
03         if (e.getSource() instanceof Timer){
04             Timer timer = (Timer) e.getSource();
05             //...
06             long callInterval = System.currentTimeMillis()-lastCall;
07             int lag = (int) ( callInterval - (long) timer.getDelay() );
08             //...
09         }
10     }
11 }
```

Fig. 8. Example of the difference between gold standard and our slicing technique for the API 'Timer.getDelay().' The underlined lines were selected by the gold standard only.

the quality of the `eXoaDocs` results R_{E}, we measured the precision and recall: $\frac{|R_{\mathrm{G}} \cap R_{\mathrm{E}}|}{|R_{\mathrm{E}}|}$ and $\frac{|R_{\mathrm{G}} \cap R_{\mathrm{E}}|}{|R_{\mathrm{G}}|}$, respectively.

Compared to the gold standard ranking, `eXoaDocs` achieved high precision and recall from five assessors: 45% and 71%, respectively.

Summarization precision/recall. Similarly, we constructed a gold standard for summarization. Using the same 20 API methods, we randomly collected the entire code from each API method and asked each human assessor to select lines of code to appear in the summary. We labeled the set of lines selected from *majority voting* as L_{G} and the lines selected from `eXoaDocs` as L_{E}. We also measured the precision and recall, that is, $\frac{|L_{\mathrm{G}} \cap L_{\mathrm{E}}|}{|L_{\mathrm{E}}|}$ and $\frac{|L_{\mathrm{G}} \cap L_{\mathrm{E}}|}{|L_{\mathrm{G}}|}$.

Compared to the summarization gold standard, `eXoaDocs` summarization also achieved both a high precision and recall, that is, 83% for precision and 70% for recall.

We analyzed the lines missed by our summarization step, as the recall was not 100%. Figure 8 shows an example of the difference between the gold standard and our slicing technique. Even though our slicing technique effectively found more relevant lines, such as lines 4 and 7, it could not find some lines that were less important but gave additional information to understand the query API method. For example, line 6 initializes the variable "callInterval" used with the query API method, but our slicing technique missed it, as it is not an argument of the query API method.

4.5. Sample Code Examples

In this section, we present some sample code examples in `eXoaDocs`.

Figure 9 shows three code examples. Figure 9(a) is an example of "Resultset.getBoolean." It contains relevant information about variables and classes of the query API method. Figures 9(b) and (c) are examples of "StringBuilder.substring." These code examples show that `eXoaDocs` also successfully presents overloaded API methods well. These overloaded API methods are detected when we analyze the source code and summarize it into code snippets by building an IAT.

5. USER STUDY

This section presents a user study conducted to evaluate how `eXoaDocs` affect real software development tasks.

```
int t_iFieldIndex = t_Query.getFieldIndex(field);
//...
ResultSet t_ResultSet = getResultSet();
//...
result = t_ResultSet.getBoolean(t_iFieldIndex);
```

(a) ResultSet.getBoolean(int columnIndex).

```
public static String substringAfter(CharSequence string,String matchString){
  if (string == null)return null;
  if (matchString == null || matchString.length() == 0)return string.toString();
  int index = indexOf(string,matchString);
  return (index < 0)? "": string.toString().substring(index + matchString.length());
}
```

(b) StringBuilder.substring(int start).

```
public static String substituteMacros(String aString, Properties someProperties){
  StringBuilder buf = new StringBuilder(aString);
  int idx = 0;
  while ((idx = buf.indexOf("${", idx)) >= 0) {
    int endIdx = buf.indexOf("}", idx + 2);
    if (endIdx > 0) {
      String key = buf.substring(idx + 2, endIdx);
      String value = someProperties.getProperty(key);
      if (value != null) {
        buf.replace(idx, endIdx + 1, value);
      }
      else {
        idx += 2;
      }
    }
  }
  return buf.toString();
}
```

(c) StringBuilder.substring(int start, int end).

Fig. 9. Samples of code examples in eXoaDocs.

5.1. Study Design

We conducted a user study with 24 subjects (undergraduate computer science students at the Pohang University of Science and Technology). For development tasks, we asked the subjects to build SQL applications using the java.sql package. The subjects were randomly divided into two groups that used either eXoaDocs or JavaDocs to complete the following tasks.

— $Task_1$. Establish a connection to the database.
— $Task_2$. Create SQL statements.
— $Task_3$. Execute the SQL statements.
— $Task_4$. Present the query results.

We compared the following metrics to measure the productivity and code quality.

The goal of our user study was to validate the presented research that automatic annotation of API methods with usage examples can (1) increase productivity, (2) improve the quality of code, and (3) reduce cognitive loads, both implicitly (based on

Table X. Averaging the Cumulative Completion Time of Only Those Who Completed the Task (min:s)

Group	Task_1	Task_2	Task_3 & Task_4
eXoaGroup	8:53	23:34	30:32
JDocGroup	14:40	25:03	32:03

automatically logged user development behaviors) and explicitly (using a post-study survey).

5.1.1. Productivity. To evaluate productivity, we used the following measures.

— *Overall task completion time* measures the overall time to finish each given task.
— *API document lookup* measures the number of lookups, which suggests the scale of development process disturbance.

5.1.2. Code Quality. We prepared a test suite that created a table, inserted two tuples into the table, and printed the tuples using the submitted subject code. Based on whether the presented tuples matched the tuples inserted in the test suite, we classified the tasks submitted by subjects as "pass" or "fail."

5.1.3. Cognitive Load. We asked the subjects to quantify the usefulness of the API documents and code examples on a scale of 1 to 5.

5.2. Participants

We first conducted a pre-study survey of the subjects to observe their prior development experiences. We enumerated example development tasks with varying degrees of difficulty and asked the subjects to choose the tasks they could perform. On the basis of the survey, we observed that none had prior development experience with the java.sql package.

We divided the subjects into two groups—"JDocGroup," referring to the group that used regular JavaDocs (the control group), and "eXoaGroup," referring to the group that used `eXoaDocs`. For a fair study, using the pre-study survey, we first categorized the subjects into four levels based on their Java expertise, because experience with using the Java language varied in general. We then randomly divided an equal number of subjects from each level into two groups. (If the subjects knew the group to which each subject belonged, it could have affected the study result.) To prevent this problem, we mirrored JavaDocs and hosted both JavaDocs and `eXoaDocs` on the same server.

5.3. Study Result

5.3.1. Productivity. We first evaluated the productivity of the two groups. We automatically logged all the document lookups carried out by all subjects. Based on the log, we measured the task completion time and the number of document lookups.

We only considered the development time of subjects who successfully completed each task. Specifically, as subjects tackled tasks in a linear order, we concluded that Task_i is completed when the subjects referred to the relevant documents for the next Task_{i+1}. To distinguish whether subjects were simply browsing the document or referring to these documents, we checked the time spent on the given document. We assumed that the user started on Task_i, only when each respective document was referred to for more than 30s.

Table X shows the average completion time of each group. The task completion times for eXoaGroup were higher than that for JDocGroup. For Task_1, eXoaGroup was up to 67% faster in terms of the average completion time. Note that the completion times for Task_3 and Task_4 are the same, since both tasks referred to the same document. This improvement was statistically significant according to a student's t-test at $P < 0.05$

Table XI. Average Number of Document Lookups

Group	Total lookups	Distinct lookups	Relevant lookups	$\frac{\text{relevant}}{\text{distinct}}$
eXoaGroup	**5.67**	**3.25**	**2.33**	**0.72**
JDocGroup	17.58	7.5	3.25	0.43

Table XII. Number of Subjects that Successfully Completed the Test Suite for Each Task

Group	Task_1	Task_2	Task_3	Task_4
eXoaGroup	**11**	**11**	3	**3**
JDocGroup	10	8	**4**	1

Note: More subjects in eXoaGroup than in JDocGroup successfully completed the tasks.

level for Task_1. Further, its 90% confidence intervals are $[4:29, 13:16]$ for eXoaGroup and $[10:44, 18:37]$ for JDocGroup.

Table XI compares the document lookup behavior of the two groups. A larger number of lookups indicates that the subjects referred to many pages to understand the correct usage. We present the total number of document lookups, the number of distinct lookups minus duplicated counts, and the number of relevant lookups counting only those on the pages that directly related to the given tasks. Lastly, $\frac{\text{relevant}}{\text{distinct}}$ indicates how many documents, among the distinct documents referred to, were actually relevant to the given task, which suggests the hit ratio of the referenced documents.

eXoaGroup referred to a significantly smaller number of documents, implying less disturbance in the software development process than JDocGroup. For instance, JDocGroup referred to overall three times more documents than did eXoaGroup. This improvement was statistically significant according to a student's t-test at $P < 0.01$ level for total lookups, and its 90% confidence intervals were $[3.57, 7.76]$ for eXoaGroup and $[11.60, 23.56]$ for JDocGroup. Meanwhile, the hit ratio was significantly higher (72%) for eXoaGroup than that of JDocGroup (43%).

5.3.2. Code Quality. Only a few subjects completed all of the tasks correctly within the specified 40 min. In JDocGroup, only one subject passed the suite of tests, while three in eXoaGroup could pass the tests. Table XII reports the number of correct submissions from each group. Generally, more subjects from eXoaGroup correctly completed each task. For instance, the ratios of subjects correctly completing Task_1, Task_2, and Task_4 in eXoaGroup to those finishing each task in JDocGroup were, respectively, 110%, 138%, and 300%. Note that not only did more subjects in eXoaGroup correctly complete each task, but they also completed it in considerably less time.

Surprisingly, for Task_3, the number of JDocGroup subjects passing the test suite was slightly higher than that of eXoaGroup. This exception is due to the specialty of JavaDocs "ResultSet" used in Task_3. The original JavaDocs included manually developed examples, which were succinct and well designed, to help developers understand "ResultSet." As a result, the subjects in both groups leveraged these examples for Task_3, which explains why the performance difference between the two groups for Task_3 was marginal. This result again confirms that the examples in the API documents are indeed very useful for software development tasks.

5.3.3. Cognitive Load. We surveyed the subjects to measure the cognitive load of the given tasks. Table XIII summarizes the subjects' responses to a post-study survey asking how useful the given API documents were. While the majority (7 out of 12) of eXoaGroup answered that the given documents were extremely useful or very useful, only

Table XIII. Post-Study Survey Result: Were the API Documents Helpful?

Group	Extremely useful	Very useful	Useful	Somewhat useful	Not useful
eXoaGroup	**3**	**4**	3	2	0
JDocGroup	**1**	**2**	5	2	2

Note: The majority in eXoaGroup answered that the given API documents were extremely useful or very useful.

Table XIV. Post-Study Survey Result: Were the Given Code Examples Helpful?

Group	Extremely useful	Very useful	Useful	Somewhat useful	Not useful
eXoaGroup	**1**	**4**	**3**	**2**	0
JDocGroup	**0**	**2**	**1**	**1**	0

Note: The majority in eXoaGroup answered that the given code examples were useful.

3 out of 12 from JDocGroup answered that the documents were useful. This indicates that `eXoaDocs` significantly enhance the usefulness of the documents.

Next, we asked whether the subjects used the code examples provided in the documents for development tasks. The vast majority in eXoaGroup (10 out of 12) answered in the affirmative, while only a few (4 out of 12) from JDocGroup used the code examples.

Table XIV shows the survey results regarding the usefulness of the given code examples. Five out of ten in eXoaGroup answered that the given example was extremely helpful or very helpful. We also received the following positive comments from the subjects in eXoaGroup.

Subject 1. "Enough examples and resources for the given task were provided in the given document."

Subject 2. "Usages of parameters and return types were very helpful."

6. DISCUSSION

6.1. Java Developer Comments

As an indicator for evaluating the usefulness of `eXoaDocs`, we released `eXoaDocs` at `http://exoa.postech.ac.kr` and sent it to professional Java developers, with a request for feedback on the usefulness of `eXoaDocs`. While the number of responses was not sufficiently large, the all feedback received on the generated `eXoaDocs` was positive.

Joel Spolsky[4]. "I think this is a *fantastic* idea. Just yesterday, I was facing this exact problem. . . the API documentation wasn't good enough, and I would have killed for a couple of examples. It sounds like a very smart idea. . . "

Developer 2. "Automatic example finding sounds really good. It would help developers significantly. In fact, I struggled many times to understand APIs without examples."

Developer 3. "API documents that have code examples are really helpful. Using them, I can reduce the development time significantly. However, JavaDocs provides

[4]Software engineer, blogger, author of *Joel on Software*, and creator of StackOverflow.

very few code examples, so I need to find additional code examples. I like MSDN because most of the methods contain code examples. For this reason, automatically finding and adding code examples seems a wonderful idea.. . . ”

6.2. User Feedback

The current eXoaDocs embed code examples selected by our organization algorithms described in Section 3. We designed three algorithms to select the most useful code examples. However, they may not work well for all API methods. They may also select code examples that are obsolete or include bugs [Kim et al. 2006]. Automatic removal of such obsolete or bug-ridden examples is a research challenge.

To address these issues, we are developing adaptive organization algorithms based on developer feedback [Joachims 2002; Radlinski and Joachims 2007; Radlinski et al. 2008], such that developers can adjust the rankings of code examples in eXoaDocs by clicking the buttons shown in Figure 2(c). Thus far, we have not received sufficient feedback from developers, since eXoaDocs was at an early stage of development at the time of writing this article. However, as we receive more feedback from developers, more important code examples will get positive feedback. This will increase the usefulness of eXoaDocs.

6.3. Source Code Repository

Currently, our eXoaDocs depend on the Koders code search engine. We generated queries from API documents, sent the queries to Koders, and collected up to 200 pieces of source code for each API method. The quality of the extracted code examples thus relies heavily on the quality of Koders search results, and this quality may not be optimal in some cases, as illustrated by the examples in Figure 1.

Carefully collecting source code from well executed projects or manually collecting code examples, such as the Apache Software Foundation [ASF 2010] or Java Examples [Java Examples 2010], and indexing them may yield higher-quality code example extractions. Building our own source code repository for eXoaDocs remains future work.

6.4. Summarization

Currently, our API slicing technique only considers related classes, arguments, and the query API method using a rule-based approach. Though this rule-based approach effectively finds relevant information, it has some limitations.

First, API methods appearing before or after the query API method are not considered in our approach, but a sequence of such API methods can be a useful feature when such API methods are order sensitive. However, for the remaining API methods that are not order sensitive, such sequence representation would lead to false positives and also incur the overhead of searching sequences, which is more expensive than searching vectors. Finding the sweet spot for this trade-off is non-trivial, which we leave as a future research direction. Second, our current approach cannot find correct type information for polymorphic methods and cannot cover some cases, such as instance variable of the class. Improving our summarization module remains future work.

6.5. Extension of IR Metrics for Evaluation

When we compared the quality of code examples between `eXoaDocs` and other code search engines (Table IX), we did not consider classical metrics, such as normalized discounted cumulative gain (nDCG) and mean reciprocal rank (MRR), because they penalize diversified results, thereby covering multiple intents, which is a key goal of our work. Evaluating diversity-aware MRR, that is, MRR–IA [Agrawal et al. 2009], would be relevant, which is defined as follows.

$$\text{MRR–IA}(Q,k) = \sum_{c} P(c|q)\text{MRR}(Q,k|c).$$

However, this would require knowledge of $P(c|q)$, which can be obtained from user feedback or `eXoaDocs` log. As we are currently collecting them, using such a log for MRR–IA evaluation will be an interesting future topic.

6.6. Popularity Bias

In addition to usage examples, eXoaDocs provide popularity information for each method (Figure 2(a)). To determine the popularity, we count the number of code examples generated using our framework. This popularity information is valuable, especially for new developers who do not know which methods to use for their tasks. However, this information can also be misleading for new API methods. While our tool can automatically include code examples as soon as they are available, the popularity for new API methods will be low until enough code examples are accumulated. This is commonly known as the "new item" problem [Balabanović and Shoham 1997] in recommendation systems. This problem can be partially addressed by introducing special marks for newly added API methods.

6.7. Threats to Validity

We identified the following threats to validity.

— *The API documents used may not be representative*. JavaDocs JDK 5 was used in this article. Since we intentionally chose the most popular and commonly used JavaDocs, we may have a document selection bias.

— *The assigned tasks may not be representative*. We assigned database related programming tasks to subjects for our case study, which may or may not be representative scenarios for using eXoaDocs.

— *The subject groups may not have been divided equally*. We divided the subjects into four levels based on a pre-study survey of prior programming experience and randomly divided each level into two groups. However, it is still possible that the programming experiences varied between the two groups. We also acknowledge that the subjects may not properly represent industrial developers.

7. RELATED WORK

This section provides an overview of related research efforts and discusses how our work is distinguished. This work is based on and significantly extends the ideas briefly sketched in Kim et al. [2009] and their formal descriptions [Kim et al. 2010], by adding Section 3 to investigate a solution space of organization algorithms. In addition, evaluations in Section 5 were extended to use a public gold standard set built by human assessors, which would enable follow-up work to compare its quality against our system.

7.1. Example Recommendation

Since code examples are frequently requested artifacts, there have been efforts to recommend good examples. For example, PHP API documents [PHP 2010] provide a large set of good code examples as user contributed notes. Alternatively, Java Examples [Java Examples 2010] and KodeJava [KodeJava 2010] recently collected user-contributed code examples. Often these manually developed and recommended code examples are of high quality, and they play an important role in understanding APIs. However, they largely rely on human effort.

To address the limitations of human effort, many automatic code recommendation systems have been proposed [Holmes and Murphy 2005; Sahavechaphan and Claypool 2006; Xie and Pei 2006; Zhong et al. 2009]. These either take specific keywords from developers or automatically generate keywords. They then query their source code repositories to retrieve useful code examples for the query keywords or working source code.

Holmes and Murphy [2005] proposed a technique that recommends source code examples from a repository by matching structures of given code. XSnippet [Sahavechaphan and Claypool 2006] provides a context-sensitive code assistant framework that provides sample source code snippets for developers.

MAPO [Xie and Pei 2006; Zhong et al. 2009] extracts frequent API method usage sequences, clusters source code with similar sequences, and presents clusters. Although this work shares similarities with our approach, we observe the following key differences. First, because not all API method calls are order sensitive, abstracting code as API method call sequences may separate two code examples with the same semantics into two clusters. Similarly, sequences with and without optional calls are divided into different clusters, which may lead to too many clusters, while our vector-based abstraction generates a single cluster in these two cases. Second, MAPO uses the API method call sequence as summarization and does not provide information on its context, such as how the API method arguments were defined and populated, while our summarization method provides both the API method call and its context.

These code recommendation approaches are similar to ours in that they use source code repositories and suggest code examples automatically. However, several differences exist. First, their systems require special development environments, such as an Eclipse plug-in, while our code examples in eXoaDocs are accessible without requiring any special tool. The second difference is that our code example presentation algorithms present only the most representative examples using clustering and ranking based on the extracted features. Lastly, unlike previous approaches requiring the API method name to suggest related examples, we provide popularity information to guide developers to the frequently used API methods, even when they do not know the names.

Similar to this popularity information, there are existing systems that identify commonly used API methods. SpotWeb [Thummalapenta and Xie 2008] detects hotspots which are commonly reused API classes and methods. Holmes and Walker developed PopCon [2007, 2008], which determines the popularity of API methods based on their calls used in existing software. However, our eXoaDocs provide both popularity information and code examples simultaneously.

7.2. Code Search

While code recommendation tools build upon development environments (e.g., Eclipse plug-ins) to consider the current development context for searches, a more general approach would be a stand-alone search engine. Commercial code search engines, such as Koders [Koders 2010] and Google Code Search [Google 2010], take keywords as queries and retrieve source code where the query keywords occur frequently. As they use keyword-based search, they usually match the query keywords with the comments in the source code. As a result, they work well for comment search. However, such comments are often sparse and not useful when developers want to find an example of specific API methods. In such a case, an API-based code recommendation framework is more effective. As a result, they fail to retrieve illustrative code examples, as shown in Section 4.3.

Meanwhile, DeMIMA [Guéhéneuc and Antoniol 2008] and DECKARD [Jiang et al. 2007] studied the semantic features of source code to tackle a different problem of searching for clones when a code segment is a query. As these approaches require a code segment as a query, we cannot use them to find examples for an API document, but we adopt their notion of semantic features.

8. CONCLUSION

In this article, we introduced a new code example recommendation system for intelligent code searches by embedding API documents with high-quality code example summaries mined from the Web. We proposed three organization algorithms that select the most representative code examples to best satisfy user intent. For JDK 5, eXoaDocs augment code examples for about 75% of methods. Compared with Koders and Google Code Search, 92% of eXoaDocs results showed proper API usage examples, while only 22% and 12% of the top query results in Koders and Google Code Search showed appropriate API method usage in their snippets. In addition, comparison with the gold standard showed that our summarization and ranking techniques are highly precise, and our user study results indicate that the code examples in eXoaDocs improve programmer productivity.

As a future direction, we will improve the system by (1) developing example ranking by incorporating user feedback, (2) enhancing summarization using more precise analysis by considering data types as well, and (3) building our own codebase to eliminate the bias incurred using Koders.

REFERENCES

Agrawal, R., Gollapudi, S., Halverson, A., and Ieong, S. 2009. Diversifying search results. In *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining (WSDM'09)*.

ASF. 2010. The Apache Software Foundation. http://www.apache.org/.

Bajracharya, S. and Lopes, C. 2009. Mining search topics from a code search engine log. In *Proceedings of the 6th Working Conference on Mining Software Repositories (MSR'09)*.

Balabanović, M. and Shoham, Y. 1997. Fab: Content-based, collaborative recommendation. *Commun. ACM*.

Campbell, N. A. 1984. Some aspects of allocation and discrimination. In *Multivariate Statistical Methods in Physical Anthropology*. Springer, Berlin. 177–192.

Devanbu, P., Karstu, S., Melo, W., and Thomas, W. 1996. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of the International Conference on Software Engineering (ICSE'96)*.

Erkan, G. and Radev, D. R. 2004. Lexrank: Graph-based lexical centrality as salience in text summarization. In *J. Artif. Intell. Res.*

Fleiss. 2010. Fleiss' kappa. http://en.wikipedia.org/wiki/Fleiss'_kappa.

Foody, G. M., Campbell, N., Trod, N., and Wood, T. 1992. Derivation and applications of probabilistic measures of class membership from the maximum likelihood classification. *Photogramm. Eng. Remote Sens. 58*, 1335–1341.

Gaffney, J. E. and Durek, T. A. 1989. Software reuse—key to enhanced productivity: Some quantitative models. *Inf. Softw. Technol.*

Google. 2010. Google Code Search. http://www.google.com/codesearch.

Guéhéneuc, Y.-G. and Antoniol, G. 2008. DeMIMA: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*

Holmes, R. and Murphy, G. C. 2005. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE'05)*.

Holmes, R. and Walker, R. J. 2007. Informing Eclipse API production and consumption. In *Proceedings of the ETX-Eclipse Technology Exchange Conference*.

Holmes, R. and Walker, R. J. 2008. A newbie's guide to eclipse APIs. In *Proceedings of the 5th Working Conference on Mining Software Repositories (MSR'08)*.

Horwitz, S., Reps, T., and Binkley, D. 1988. Interprocedural slicing using dependence graphs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'88)*.

Jaccard, P. 1901. E'tude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Socie'te' Vaudoise des Sciences Naturelles*.

Java Examples. 2010. Example source code. http://java2s.com/.

Java2Xml. 2010. Java2XML Project Home Page. `https://java2xml.dev.java.net/`.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'02)*.

Kim, J., Lee, S., Hwang, S.-W., and Kim, S. 2009. Adding examples into java documents. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09)*.

Kim, J., Lee, S., Hwang, S.-W., and Kim, S. 2010. Towards an intelligent code search engine. In *Proceedings of the National Conference of the American Association for Artificial Intelligence (AAAI'10)*.

Kim, S., Pan, K., and Whitehead, Jr., E. E. J. 2006. Memories of bug fixes. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

KodeJava. 2010. Learn Java Programming by Examples. `http://www.kodejava.org/`.

Koders. 2010. Open Source Code Search Engine. `http://www.koders.com`.

Leopard. 2010. Leopard Reference Library. `http://developer.apple.com/referencelibrary/index.html`.

Lim, W. C. 1994. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*

MSDN. 2010. MSDN Library. `http://msdn.microsoft.com/en-us/library/default.aspx`.

Page, L., Brin, S., Motwani, R., and Winograd, T. 1999. The pagerank citation ranking: Bringing order to the web. Tech. rep.

Pelleg, D. and Moore, A. 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the International Conference on Machine Learning (ICML'00)*.

PHP. 2010. Hypertext Preprocessor. `http://www.php.net`.

Radlinski, F. and Joachims, T. 2007. Active exploration for learning rankings from clickthrough data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'07)*.

Radlinski, F., Kurup, M., and Joachims, T. 2008. How does clickthrough data reflect retrieval quality? In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM'08)*.

Sahavechaphan, N. and Claypool, K. T. 2006. XSnippet: Mining for sample code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*.

Thummalapenta, S. and Xie, T. 2008. SpotWeb: Detecting framework hotspots via mining open source repositories on the web. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'08)*.

Weiser, M. 1981. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE'81)*.

Xie, T. and Pei, J. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'06)*.

Xu, R. and Wunsch, I. 2005. Survey of clustering algorithms. *IEEE Trans. Neural Networks*.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. 2009. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'09)*.

Received June 2011; revised March, July 2012; accepted September 2012