Towards an Intelligent Code Search Engine

Jinhan Kim, Sanghoon Lee, Seung-won Hwang
Pohang University of Science and Technology
{wlsgks08,sanghoon,swhwang}@postech.edu

Sunghun Kim
Hong Kong University of Science and Technology
hunkim@cse.ust.hk

Abstract

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 the API usages, while searching for good code examples requires non-trivial effort.

To address this problem, we propose a novel code search engine, combining the strength of browsing documents and searching for code examples, by returning 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.

1. Introduction

As reusing existing Application Programming Interfaces (APIs) significantly improves programmer productivity and software quality (Gaffney and Durek 1989; Devanbu et al. 1996; Lim 1994), many developers are searching for API information on the Web, such as API documents or API usage examples, to understand the right usage of APIs.

Specifically, developers may not know which API to use, in which case they need to "browse" API documents to read the descriptions and select the appropriate one. Alternatively, developers may know which API 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, when developers are not sure of which API to use, API creators usually provide documents, and developers read the API descriptions, written in a human readable language, to teach themselves how to use the available APIs and select the proper one. However, as textual descriptions are often ambiguous and misleading, developers are reported to often combine browsing with *searching* for code examples to clarify results (Bajracharya and Lopes 2009).

To address these *searching* needs, developers often use several of the commercial search engines launched recently, including Koders (Koders 2009) and Google Code Search (Google 2009), using the API name as a query keyword to find good code examples. However, search results from Koders, shown in Figure 1, do not always meet the developers' expectations, as the snippets of both top search results show matches in the comments of source codes 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 examples.

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.
```

Figure 1: Koders Top-2 results when the query is "Connection prepareStatement"

To reduce this effort, some API documents such as the MSDN from Microsoft and the Leopard Reference Library from Apple include a rich set of usage code examples, so that developers do not need to find additional examples using search engines. However, as manually crafting high-quality code examples for all APIs is time consuming, most API documents lack code examples. To illustrate, JDK 5 documents include about 27,000 methods, but only about 500 of them (around 2%) are explained with code examples.

One basic approach would be to include code search engine results (code examples) in advance leveraging existing code search engines to embed the top results as code examples. However, these search engines, which treat codes as simple text, fail to retrieve high-quality code examples, as demonstrated in Section 3.2. Similarly, one may consider, code recommendation approaches proposed (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 to provide suitable code examples.

In clear contrast, we propose an intelligent code search engine that searches, summarizes, and embeds the necessary information in advance by automatically augmented with high-quality code examples, as Figure 2 illustrates. Each API is annotated with popularity, represented by a bar in

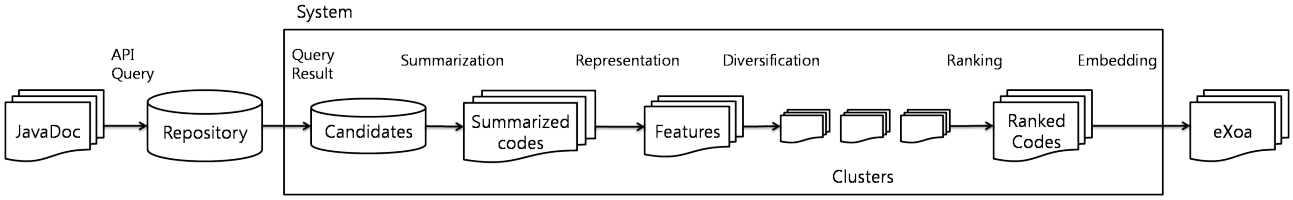

Figure 3: Process of Automatic Example Oriented API Document Generation

Figure 2: An example page of generated eXoaDocs. (a) The popularity information of methods (b) Code examples (c) User's feedback button

Figure 2 (a), to help developers' browsing needs to find APIs that are used frequently in programming tasks. For each API, two to five code examples are presented, as Figure 2(b) shows. From this point on, we call these documents *example oriented API Documents* (*eXoaDocs*).

Specifically, to ensure that code examples embedded in the documents effectively summarize diverse API usages used in real-life code corpus, we adopt the intuition of multi-document summarization. This refers to grouping examples into clusters and assessing their centrality to extract representative code examples, pioneered by (Erkan and Radev 2004) and built upon by many other systems. Our evaluation results show that our approach summarizes and ranks code examples with high precision and recall. In addition, our case study shows that using eXoaDocs boosts programmer productivity and code quality.

2. Our Approach

Our framework consists of four modules, summarization, representation, diversification, and ranking (Figure 3).

First, our framework builds a repository of candidate code examples by leveraging an existing code search engine and summarizing them into effective snippets (*Summarization*). After that, we extract semantic features from each summarized code example for clustering and ranking (*Representation*). Next, we cluster the code examples into different usage types based on semantic features to present diverse usage types to developers (*Diversification*). Then, we rank the code examples in each cluster to present the most representative code example from each cluster (*Ranking*). Finally, we embed the most representative code example from each cluster into the API documents and generate eXoaDocs.

2.1 Summarization

The first module of our framework searches and collects potential code examples for each API.

To achieve this, we leverage a code search engine, Koders, by querying the engine with the given API name and the interface name extracted from API documents, and collect the top-200 codes for each API. We decide 200 as the retrieval size, based on our observation that most results ranked outside the top-200 are irrelevant to the query API or redundant.

```
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 }
```

Figure 4: A manually crafted code example in JavaDocs

Next, we summarize the codes into *good example snippets* that best explain the usage of the given API. To obtain initial ideas on defining good example snippets, we first observed manually crafted 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.

- Good example snippets should include the actual API, *e.g.*, "f.add()" in Figure 4, and its semantic context, such as how to declare an argument "ex1" within the method.
- Irrelevant code, regardless of the textual proximity to the API, can be omitted as in line 5 in the manually written example.

Recall that, both of the top-2 code snippets from Koders in Figure 1, violate all characteristics needed to be good example snippets, by (1) failing to show the actual API and (2) summarizing based on the text proximity to keyword matches.

In a clear contrast, we achieve summarization based on the semantic context, by following the steps below.

• **Method extraction**: We first identify the methods that contain the given API because they show the usage and context of the given API.

• **API slicing**: Second, we extract only the *semantically relevant* lines for the given API using slicing techniques.

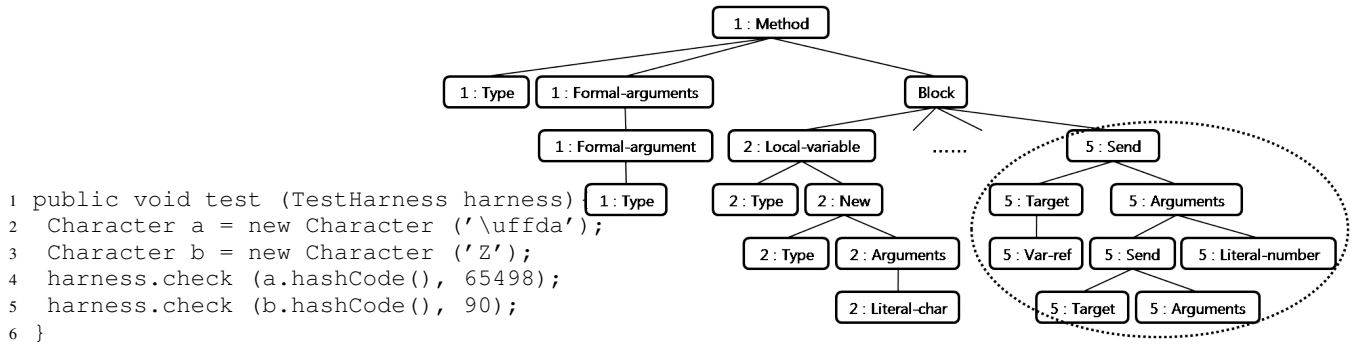

Figure 5: eXoaDocs code example of "Character.hashcode()" and its abstract syntax tree

Relevant lines satisfy at least one of the following requirements: (R1) declaring the input arguments for the given API, (R2) changing the values of the input arguments for the given API, (R3) declaring the class of the given API, or (R4) calling the given API. For example, in Figure 4, line 2 is relevant because it declares the Frame class of the API (R3), line 3 because it declares the input argument (R1), line 4 because it initializes the input argument (R2), and line 6 because it calls the given API (R4). All other irrelevant lines were omitted.

To identify 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 2009). To illustrate, Figure 5 shows a code example and its AST. Each node of the AST represents an element of the Java source code. For example, the *Formal-Argument* represents the input arguments of the method, "harness", in line 1.

Table 1: An Example of an Intra-method Analysis Table for the code in Figure 5

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

Next, we find the relevant lines from the AST. To identify the relevant lines, we traverse the AST, check the name and type information of all variables and API calls, and build a simple lookup table, Intra-method Analysis Table (IAT).

Each row in the IAT corresponds to an API and consists of four parts, as Table 1 shows: the line of the API, name and line information of the class of the API, the API name, and the type and line information of the argument list of the API (for simplicity, we omit the rows on the "harness.check()"). For example, the first row of Table 1 shows that (a) "hashCode()" is called at line 4, (b) its class type is "Character" with name "a" (declared at line 2), and (c) "hashCode()" has no argument.

2.2 Representation

We now discuss how to represent code examples for clustering and ranking. One extreme is to treat code examples as simple texts. Another extreme is to represent code examples as ASTs and compute similarity between the ASTs. The former is highly efficient but neglects the semantic context, while the latter is expensive but considers the semantic context.

We achieve a balance between these two extremes, by extracting element vectors from the ASTs, inspired by a clone detection algorithm, DECKARD (Jiang et al. 2007), and computing similarity using vector comparison. Specifically, we consider the elements of the Java source code that represent the characteristics of code examples. The Java source code can be divided into several small element parts such as *if statement* and *variable declaration*.

DECKARD proposes *q-level characteristic vectors* to approximate the semantic features of the parse tree for each code, and *q-level characteristic vectors* are used for similarity checking between codes where q is the depth of the parse tree. *q-level characteristic vectors* consist of an n-dimensional numeric vector $< c_1, ..., c_n >$ where n is the number of elements in the parse tree such as loops, and each c_i is the count of occurrences of a specific element. DECKARD manually judges and selects "relevant" elements. For example, when computing vectors for the subtree for the code in line 5, *i.e.*, the tree segment in the dotted circle in Figure 5, the characteristic vector for the selected elements *arguments*, *local-variable*, and *new* is $< 2, 0, 0 >$.

Similarly, we use 85 Java source code elements, detected from ASTs using the "java2xml". (For the entire list of all 85 elements, refer to (Java2Xml 2009)). In addition, we consider two more features specific to our problem context: the frequency of the given API calls and the example size. Overall, we use 87 element vectors to represent code examples.

2.3 Diversification

Our next step is to diversify the code examples to identify the different usage types. Since API examples show various kinds of situations, it is desirable to show all diverse usage types of code examples rather than presenting one type of examples. We cluster the code examples based on their extracted vectors described in Section 2.2.

One issue in clustering is choosing the number of clusters k. In our problem, as the number of examples that can be shown for each API is usually very small (as documents would be unnecessarily long otherwise), we consider k in a bounded range, *e.g.*, $k = 2$ to 5. That is, we simply invoke the k-means algorithm four times and pick the result with the best quality using the quality metrics we defined as follows.

• **Centroid distribution:** Clustering where centroids are distributed evenly suggests a good coverage of varying usage types. In contrast, clusters with centroids skewed in certain areas are likely to cover the same type of redundancy. For this reason, we quantify the quality as $\frac{1}{var_i}$ for variance var_i of centroids from the clustering results when $k = i$.

• **Sum of the squared error (SSE):** We compute the error of each vector, *i.e.*, its distance to the centroid, and SSE. Clustering with the smallest SSE suggests that centroids are statistically better representatives of all vectors in the cluster and thus indicates high quality clustering results. We quantify the quality as Δ_i, which represents the difference in SSE when $k = i$ and $k = i - 1$.

• **Hierarchical clustering:** The usage type clusters typically represent a hierarchical structure. That is, a cluster can be split into sub-types (the lower layers in the hierarchy) or two clusters can be merged to represent their super-type (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 $k = i$) come from the same cluster in the results when $k = i - 1$, and 0 otherwise.

We choose a k value which yields the highest sub-score and k determines the number of usage types for each API.

2.4 Ranking

Code example ranking is a crucial part of eXoaDocs for providing better search results, since the number of examples we can show is very limited and we cannot afford to pick bad examples. We now discuss how to select one distinctive and representative code example from each cluster (intra-cluster ranking) and then rank the selected representative code examples (inter-cluster ranking).

First, we describe intra-cluster ranking, ordering the code examples in one cluster, by aggregating the following three factors established in data mining and software engineering literature. We automatically learn the aggregation function using user feedback collected in Figure 2 (c), as similarly studied in (Chapelle and Keerthi 2009).

• **Representativeness:** Code examples in each cluster can be ranked by the degree of membership, typically measured as the similarity with a representative of the cluster, *e.g.*, the centroid, using the *L1* distance for similarity. Specifically, the measure of the representativeness is $\min(\frac{1}{similarity}, 1)$.

• **Conciseness:** Usually developers prefer concise examples rather than lengthy ones. Therefore, concise examples, *i.e.*, those with fewer lines of code, are highly ranked. We use $\frac{1}{\text{length}}$ to measure the conciseness.

• **Correctness:** APIs with the same name can belong to different classes or can take different arguments. In addition, the IAT may miss finding a correct type of class or argument. If the given code uses an API that has the right class or the matching of arguments, it should be ranked higher.

Second, we present the inter-cluster ranking of ordering representative code examples, representing the different usage types identified, to reflect the popularity of each usage type. In our implementation, we count the number of code examples in each cluster to quantify the popularity.

3. Evaluation

We evaluated the quality of our search results, compared with existing document APIs, code search engines, and golden standard results. The search results used for this evaluation are open to the public at `http://exoa.postech.ac.kr/`.

3.1 Comparison with JavaDocs

We compared the quantity of examples that eXoaDocs generated for the JDK 5, with JavaDocs. Out of more than 20,000 methods, eXoaDocs augmented examples for more than 75% while only 2% were augmented in JavaDocs.

3.2 Comparison with Code Search Engines

We queried with 10 randomly selected APIs and manually categorized the top-k results (where k is between 4 and 5, which 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 below.

• **Relevant code and snippet:** The code snippet presented includes the appropriate API and all parameters used in the API are well explained in the code example snippet (*i.e.*, a good *ranking* and good *summarization*).

• **Relevant code:** The code includes the correct API usage but is incorrectly summarized and fails to show and highlight the API usage in the snippet (*i.e.*, a good *ranking* but bad *summarization*).

• **Irrelevant code:** In this case, the queried API does not appear in both the code and its summarized snippet (*i.e.*, a bad *ranking* and bad *summarization*).

The vast majority of Koders and Google Code Search results, 70% and 78% respectively, were irrelevant, showing the comment lines or import statements that are not helpful. This lack of relevance is explained by the fact that, when building on textual features, they cannot distinguish between relevant and irrelevant matches. On the other hand, 92% of the code snippets in eXoaDocs showed proper API usage in the summary, while only 22% and 12% of the snippets in Koders and Google Code Search satisfied the requirements.

We evaluated eXoaDocs with respect to *diversification*. Among the few relevant snippets from Koders (22%) and Google Code Search (12%), 30.8% and 16.7% were duplicate code examples. In contrast, the code examples in eXoaDocs covered diverse usage types and included only 8.7% duplicates.

3.3 Comparison with Golden Standard

We evaluated the ranking and summarization of eXoaDocs using golden standards built by two human assessors.

Ranking precision/recall measures: To construct golden standard top-k results, we present $2k$ random example summaries by human assessors, including the top-k summaries selected by eXoaDocs. We ask two human assessors to mark the k best examples, which are marked as golden standard results R_{G}.

To measure the quality of the eXoaDocs results R_{E}, we measure the precision and recall: $\frac{|R_{\text{G}} \cap R_{\text{E}}|}{|R_{\text{G}}|}$ and $\frac{|R_{\text{G}} \cap R_{\text{E}}|}{|R_{\text{E}}|}$ re-

spectively. However, in this case, as $|R_{\mathrm{E}}| = |R_{\mathrm{G}}|$, the precision is the same as recall.

Summarization precision/recall measures: Similarly, we constructed a golden standard for summarization, by asking human assessors to select lines of codes to appear in the summary. We denoted a set of selected lines as L_{G} and the lines selected by eXoaDocs as L_{E}. We also measured the precision and recall, *i.e.*, $\frac{|L_{\mathrm{G}} \cap L_{\mathrm{E}}|}{|L_{\mathrm{G}}|}$ and $\frac{|L_{\mathrm{G}} \cap L_{\mathrm{E}}|}{|L_{\mathrm{E}}|}$.

Results: We randomly selected 20 APIs and presented $2k$ results to two human assessors to select the k golden standards. In comparison to the golden standard, eXoaDocs achieved significantly high precision and recall from both assessors, 66% and 60%. For the same 20 APIs, we presented the entire code results of eXoaDocs to assessors and asked them to obtain golden standard summarization. Compared to this golden standard, eXoaDocs summarization also achieved high precision and recall from both, where the averages were 82% for precision and 73% for recall.

3.4 Java Developer Feedback

As one indicator to evaluate 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 the documents. While the number of responses was not large enough, we had received generally positive feedback on the generated eXoaDocs.

Joel Spolsky[1]: "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 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..."

4. Case Study

This section presents our case study for evaluating how eXoaDocs affect the software development process.

4.1 Study Design

We conducted an user study with 24 subjects (undergraduate students at Pohang University of Science and Technology). For development tasks, we assigned subjects to build SQL applications using the java.sql package. Subjects were randomly divided into two groups to use either eXoaDocs or JavaDocs, to complete the 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.

Productivity

- **Overall development time:** We measured the overall development time until the completion of each given task.
- **API document lookups:** We implicitly measured the number of lookups, which suggests the scale of development process disturbance.

Code Quality We prepared a test suite creating a table, inserting two tuples, and printing these tuples, using the submitted subject codes. Based on whether the tuples presented matches to the tuples inserted in the test suite, we categorized the tasks submitted by subjects as 'pass' or 'fail'.

4.2 Participants

We divided subjects into two groups – the "JDocGroup", referring to the group that uses regular JavaDocs (the control group) and the "eXoaGroup", referring to the group that uses eXoaDocs. We mirrored JavaDocs and hosted both JavaDocs and eXoaDocs on the same server, so as not to reveal to the participants the group to which each subject belonged to, which could have affected the study result. For a fair study, using the pre-study survey, we first categorized the subjects into four levels, based on their Java expertise, then divided subjects in each level randomly in half.

4.3 Study Result

Productivity We first evaluated the productivity of the two groups. We automatically logged all document lookups of all subjects by recording when each subject accesses which document. Based on the log, we measured the task completion time and the number of document lookups.

Group	Task$_1$	Task$_2$	Task$_3$ & Task$_4$
eXoaGroup	8:53	23:34	30:32
JDocGroup	14:40	25:03	32:03

Table 2: Averaging the cumulative completion time of only those who completed the task (min:sec).

We only considered the development time of subjects that completed each task. Specifically, as subjects tackled tasks in a linear order, we concluded that a Task$_i$ is completed when the subjects "refers" to the relevant documents for the next one Task$_{i+1}$. To distinguish whether subjects simply "browse" the document or "refer" to it, we checked the time spent on the given document. We assume that the user has started on Task$_i$, only when each respective document is referred to for more than 30 seconds.

Table 2 shows the average completion time. The task completion times of the eXoaGroup were faster than those of the JDocGroup. Note that the completion time for Task$_3$ and Task$_4$ are the same, since both tasks refer to the same document. For Task$_1$, the eXoaGroup increased the development speed by up to 67% in terms of the average completion time.

Table 3 compares the document lookup behavior of the two groups. A larger number of lookups indicates that the

[1]Software engineer, blogger, and the author of "Joel on Software."

Table 3: The 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

subjects refer to many pages to figure out the usage. We present the total number of document lookups, the number of distinct lookups removing 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, are actually relevant to the given task, which suggests the "hit ratio" of the referred documents.

The eXoaGroup referred to a significantly fewer number of documents, *i.e.*, less disturbance in the software development, compared to the JDocGroup. For instance, the JDocGroup referred to three times more documents overall, than that of the eXoaGroup. This improvement was statistically significant (e.g, a p-value$= 0.004$ for total lookups). Meanwhile, the hit ratio was significantly higher (72%) for the eXoaGroup, compared to the JDocGroup (43%).

Table 4: The number of subjects passing the test suite for each task. More subjects in eXoaGroup finished tasks

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

Code Quality Only a few subjects completed all tasks correctly within the specified 40 minutes. In the JDocGroup, only one passed the test suite, while three in the eXoaGroup passed the test. Table 4 reports on how many submissions were correct from each group. Generally, more subjects from the eXoaGroup finished each task. For instance, the ratio of subjects finishing Task_1, Task_2, and Task_4 in the eXoaGroup were 110%, 138%, and 300% of those finishing each task respectively in the JDocGroup. Note that not only did more subjects in the eXoaGroup complete each task, but they also completed it in much less time.

Surprisingly, the JDocGroup performed slightly better for Task_3. This exception is due to the speciality of the JavaDocs "ResultSet" needed to solve Task_3. It includes manually crafted examples, which are succinct and well designed to help developers understand the "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. Conclusion

In this paper, we introduced a new search output system for intelligent code search, by embedding API documents with high-quality code example summaries, mined from the Web. Our evaluation and case study showed that our summarization and ranking techniques are highly precise, which improves 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 considering data types as well, and (3) building our own code-base to eliminate the bias of using Koders.

Automatically generated eXoaDocs are available at `http://exoa.postech.ac.kr/`.

6. Acknowledgments

Our thanks also goes to Joel Spolsky and the Java developers for their feedback on eXoaDocs. This research was supported by National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF), grant number R11- 2008-007-03003-0.

References

Bajracharya, S., and Lopes, C. 2009. Mining Search Topics from a Code Search Engine Log. In *MSR '09*.

Chapelle, O., and Keerthi, S. S. 2009. RankSVM: Efficient Algorithms for Ranking with SVMs. In *Information Retrieval '09*.

Devanbu, P.; Karstu, S.; Melo, W.; and Thomas, W. 1996. Analytical and empirical evaluation of software reuse metrics. In *ICSE '96*.

Erkan, G., and Radev, D. R. 2004. Lexrank: Graph-based lexical centrality as salience in text summarization. In *Journal of Artificial Intelligence Research '04*.

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

Google. 2009. Google Code Search. `http://www.google.com/codesearch`.

Holmes, R., and Murphy, G. C. 2005. Using structural context to recommend source code examples. In *ICSE '05*.

Java2Xml. 2009. 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 *ICSE '07*.

Koders. 2009. Open Source Code Search Engine. `http://www.koders.com`.

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

Sahavechaphan, N., and Claypool, K. T. 2006. XSnippet: mining for sample code. In *OOPSLA '06*.

Zhong, H.; Xie, T.; Zhang, L.; Pei, J.; and Mei, H. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009*.