Automatically Finding Web Documents Related to a Code Sample

Phan The Dai[†] and Katsuhisa Maruyama[††]

Code search engines on Web have played important roles for novice programmers during developing software or studying programming by reference examples. However, they have faced a problem from search results because these code samples lack context or can be unfamiliar code snippets that are incomprehensible. They must find external information resources by using another Web search engine with numerous queries manually and too many pages can be visited. This paper presents a method to automatically find related Web documents to support them to comprehend source code. We utilize Web search service APIs for collecting related information of Web pages to process. Advantaged techniques of information retrieval are used to rank the text documents. For demonstrating benefits of this method, the proposed method is applied into an architecture model of a source code search engine and its implementation. Experimental results from several case studies showed that the method is useful in supporting programmers understanding new concepts in code samples.

1. Introduction

For programmers, Internet Search has been a frequent and necessary activity to support not only learning programming but also finding solutions when they are facing with a programming problem. In which, code-specific search engines such as Google Code Search [1] can serve as powerful resources of open source code. However, they have not fitted completely the programmers' needs. Their shortcoming came from that they were designed to specifically support programmers searching for raw code samples that lack their contexts. Novice programmers often face with an unfamiliar code sample, which either declares type names referring to other packages or have a complex program structure. They must involve another general-purpose search engine including Google or Yahoo for searching documentation resources, such as Javadoc pages explaining class members in a hierarchy structure or pages containing similar code snippets with explanatory texts. This search costs the programmers more time and effort because they would prepare numerous queries manually and visit too many pages. Tool support that alleviates this kind of problem during code searching is essential.

With the above motivation, we consider that the current existing source code engines should be integrated an extra function that allows automatically recommending more external documentation for their code search results. In this paper, we propose a novel method to automatically retrieve Web documents related to code a programmer wants to understand. This method uses Web search service Application Programming Interfaces (APIs) to collect the top of ranked web search results that might explain the code sample. The collected Web documents are ranked against the code sample by using Information Retrieval (IR) techniques. In addition, to demonstrate usage benefits of the method, we present an architecture example of a source code search engine that utilizes this approach to automatically recommend documentation for each its code search results.

The main contributions of this paper are that it 1) introduces our method, named FWD, that automatically finds the related documentation on Web for source code, 2) presents a usage of the method by two models of code search engine that can recommend related Web documents for its results, and 3) provides an implementation of specific Web-based tool that realizes our method, and 4) shows the benefits of this approach through some case studies.

The remainder of the paper is organized as follows: Next, Sect. 2 presents Web search services and several IR techniques used in them. Sect. 3 describes our FWD method to find Web documents associated with a given piece of code. Sect.4 presents usage of the method through integrating this solution into a source code search engine. Sect.5 provides implementation of the specific Web-based tool. In Sect.6, we validate the efficient of this implementation through some case studies. Sect. 7 discusses several related studies. Finally, Sect. 8 gives concluding remarks and some directions for future work.

2. Background

This section starts with introducing several code search engines and then describes Web search service APIs in our automated data collection process. It also explains several common IR models that have been applied successfully into retrieving links between code and software documentation in software engineering.

2.1 Source code search engines

A number of code search tools have been proposed and developed with several alternative solutions to find information of code samples. They are based primarily around traditional IR techniques that accept queries as keywords (names of classes, names of methods, parameters, etc). These tools can support developers search on Integrated Development Environment (IDE), in the same project, Intranet, and Internet. Some significant contributions both from academic and the industry are Google Code Search [1], Koders [2], Sourcerer [4], SPARS-J

[†] Graduate School of Science and Engineering, Ritsumeikan University
[††] Department of Computer Science, Ritsumeikan University

[5], Codase [17], Code conjurer [18], and Parseweb [20].

These tools either allows search within their own indices or are built on top of the external search engines. For example, Google Code Search crawls publicly accessible source code resources including archives, CVS repositories and Subversion repositories. PARSEWeb currently uses Google Code Search to find and download likely examples of object instantiation. Sourcerer is a search engine for open source code search and provides an infrastructure for large-scale indexing. Code conjurer is a test-driven-based code search tool that is supported by a Merobase component finder. Strathcona uses developer's current structure context to recommend source code examples from its repository.

2.2 Web search service APIs

Web search engines play important roles in finding and accessing information on the Internet. General search engines allow users to search for various kinds of web content. Several current Web search services can provide both Web User Interfaces (WUIs) and APIs to access their indexes. They offer users with alternative ways to collect data as manually submitting queries to the their web user interface, to create programs that automate the task of submitting queries, or to use web search service APIs to access their index and collect data. However, several search engines have legal restrictions against automating data collection using the WUIs. Therefore, in this research, we chose using APIs for collecting information on the Web. Next, we explain the popular Web Search APIs provided by Google and Yahoo.

Google AJAX Search API [15] provides JavaScript libraries for JavaScript environments. For other Non-JavaScript environments, the API exposes a raw RESTful interface. REST is an acronym standing for Representational State Transfer. We can interact easily with the RESTful service by creating HTTP requests and processing HTTP responses. Using an API key for the Google AJAX Search API in the application (site) is completely optional but Google suggests users that they should have one.

Yahoo! Web Search API [16] is a RESTful service. Access is rate limited based on the caller's IP address, and queries are limited per IP and per day. With Yahoo! Web Search Service API, we can create HTTP requests and process HTTP responses by hand, or by using the client libraries. To support for developers, libraries for many programming languages such as Perl, Python and PHP, Java, JavaScript, and Flash and example code have been bundled together as a Software Development Kit.

2.3 IR techniques for similarity calculation between document and code

IR techniques have proven useful in many areas, for retrieving not only free-text documents but also specific structured data such as source code. Almost all current search implementations are based on or extensions of IR methods. Recently, IR techniques have been applied on retrieving links between code and software documentation such as user manuals, requirements, design documents, test cases, etc and vice versa. A widely used approach is ranked retrieval, which ranks the documents against the queries constructed from source code and returns a ranked list of documents. First, documentation and source code need to be preprocessed, such as retrieval items extraction, transforming all capital letters into lower case letters, removing stop words and stemming (reducing words to a root form). Second, similarity is computed by applying some IR models such as Probabilistic Model (PM), Vector Space Model (VSM) and Latent Semantic Indexing (LSI). As a result, a list of ranked documentations is generated. Finally, the results with the relevance degrees above the threshold value are selected from this list.

In the PM model, documents are ranked according to the probability of being relevant to a query computed on a statistical basic. Suppose that there are N documents, let d denotes for *i-th* document and query Q represents a piece of source code. Then, similarity-scoring formula between d and Q is:

$$Similarity(Q,d) = \Pr(d \mid Q) = \frac{\Pr(Q \mid d)P(d)}{\Pr(Q)}$$

For a given piece of code Q, $Pr(Q)$ is a constant and we can further simplify the model by assuming that all system documents have the same probability. Therefore, for each Q, all documents d are ranked by the conditional probabilities $Pr(Q/d)$ that can be computed by estimating a stochastic language model for each document d [7].

In VSM model, documents and queries are represented as vectors in a multi-dimensional Euclidean space where each axis corresponds to a separate term (word). The co-ordinate along the axis is a weight determined by statistical occurrence data for the term. Terms definitions are depend on applications. In our case, terms are words are extracted from the documents themselves. Once documents and queries are encoded in vectors, similarities between a document and a document or a document and a query can be deduced according to vector arithmetic (e.g. a distance function). In a widely used approach, distance between vectors is captured by the cosine of the angle between them.

Assume that, M is number of dimension of space, as the size of the vocabulary. The weight vector for document d is: $V_d = [w_{1,d}, w_{2,d}, \ldots, w_{M,d}]$. The *i-th* element $w_{i,d}$ is a measure of the weight of the *i-th* term of the vocabulary in the document d. Similarly, the weight vector for query Q is: $V_Q = [w_{1,Q}, w_{2,Q}, \ldots, w_{M,Q}]$.

Different ways of computing term weights (term weighting schemes) have been developed. One of the schemes often used in information retrieval and text mining is *term frequency–inverse document frequency* weighting, *tf-idf*. According to this weight, the importance increases proportionally to the number of times a term appears in the document but is offset by the frequency of the term in the documents collection. Term weight of term *t*

in a document d is defined as follow:

$$w_{t,d} = tf_t * idf_t$$

where tf_t is term frequency (term counts) a term t occurs in a document d; df_t is document frequency or number of documents containing term t; idf_t is inverse document frequency, which is calculated by $log(D/df_t)$.

The cosine similarity between query Q and document d can be computed by cosine similarity formula as follows.

$$Similarity(Q,d) = \cos(\theta) = \frac{V_Q \bullet V_d}{|V_Q||V_d|} = \frac{\sum_{i=1}^{M} w_{i,d} w_{i,Q}}{\sqrt{\sum_{i=1}^{M} w_{i,d}^2} \sqrt{\sum_{i=1}^{M} w_{i,Q}^2}}$$

For N documents, those documents can be represented as a $M \times N$ term-document matrix, those rows are M terms and N is the number of rows (each row is a document).

LSI is an algebraic model based on VSM. The basic assumption of LSI model is documents come from a mix of orthogonal topics; in the other words, there are some implicit relationships among the words of documents. Firstly, documents are represented as a large term-document matrix. Next, the VSM space is truncated and transformed to LSI subspace by applying singular value decomposition (VSD) to the term-document matrix. Finally, we can compute the similarity by using cosine formula in LSI subspace and filter result list according to a predetermined threshold.

3. FWD Approach: Finding Related Web Documents for a Code Sample

This FWD (Finding Web Documents) approach is based on two assumptions that are 1) several documents related to source code are possibly published on Internet, and 2) the code and its documents are often likely to share common terms. Input of the entire process in the FWD approach is a code sample (or a source file) and output is a list of related Web pages that are ranked according to relevance against this sample. The relevant Web pages should be retrieved are pages contain similar source code, API documents (Javadoc pages), pages contain code snippet and explanatory texts, tutorial pages, pages from mail listings/group/forums, pages about other similar solutions (other libraries), etc.

This process includes two main phases: phase 1 is “Collecting Documents” and phase 2 is “Ranking”, as shown in Figure 1. The sub-processes in both phases are organized in pipeline architecture; the output from phase 1 constitutes the input for phase 2.

In the phase 1, the process firstly extracts identifiers (declared names) in the code sample such as package declaration, import declarations, type declarations, super class, super interfaces, field declarations, method declarations. It constructs queries each of which can be an identifier or a combination of ones by Boolean operators. The combination purposes to improve the quality of search results. An example is as “*package name* AND *class name*”. It calls Web search service APIs such as Google AJAX Search API or Yahoo! Web Search Service API for these queries and retrieves their web search results. It obtains the addresses of these results to download web pages and parses them to extract text documents.

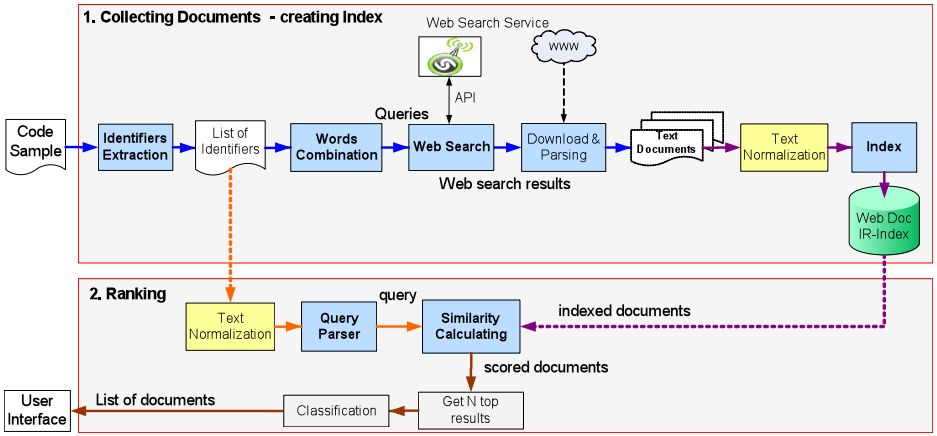

Figure 1: Overview of FWD approach

The extracted documents will be normalized as texts and indexed based on a vocabulary that is extracted from the documents them selves. Text normalization process includes the following steps: 1) transforming all capital letters into lower case letters, 2) removing stop-words or common words (such as articles, punctuation, numbers), and (3) reducing words to a root form (stemming). Text processing is required to reduce noise and to improve the semantic similarity between documents. This process performs stemming of the words in order to be able to compare words with the same stem, but different suffixes. In this research, to support for indexing process, we used Lucene that implemented the extended Vector Space Model. When text documents are indexed, inverted index structure data is created, and term weights are calculated. Similar to VSM, Lucene is using *Tf-idf* weights because *Tf-idf* values are believed to produce search results of high quality.

For ranking the text documents against the code sample, the ranking process utilizes the list of the identifiers extracted in the phase 1. In the phase2, the process normalizes these identifies as three steps for document indexing in the phase 1. Then, it inputs the results into a Query

Parser that parses them and builds a query. A scoring algorithm will calculate scores for all the indexed documents in the repositories against the above query. Finally, we select the first N results in the ranked list; apply some other processes such as classification, and presenting the final results to the user interface.

4. New Architecture Model of Code Search Engine

This section presents a usage of the FWD method by presenting models of code search engine that can recommend related Web documents for its search results. We describe two kinds of model. The first one is applied in the current source code search engines as an enterprise application in practical. The second is implemented in our research as tool aiming to validate the benefits of our approach and validate the quality of results. The main difference between two models is that the first model needs preprocesses to create an local indexed documents repository.

According to the first model, we consider that the current source code engines should be refined as an architecture shown in Figure 2. In this model, in addition to the current its code index, a search engine has one more local index repository that stores indexed web pages (documentation). To collect Web documentation and store them into the Web Doc Index, we use proposed steps in phase 1 of our FWD approach as preprocesses that need to be done before users can search. Loop over all code samples in the code repository (we assume that code index also store code samples), use information in each code sample to collect Web documents and index them. Once Web Doc index was created, finding relevant documents for each code search result is easy by using online processes of phase 2 in our FWD approach. Noting that, in this case, Ranking is responsible for as a Documents Searcher. This model has several advantages such as allowing searching quickly and presenting concurrently many results to the user interface, but it requires us resources and time to prepare data.

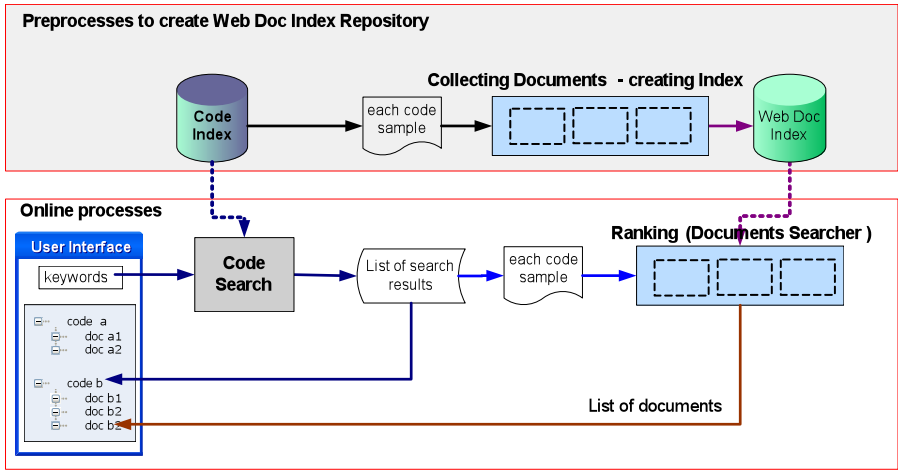

Figure 2: Enterprise architecture model using FWD approach

As mentioned in Sect.2, several code search services such as Google Code Search and Sourcerer provide free available APIs to allow accessing their index. API allows us implement search applications with the same its search quality and functions. Basing on this consideration, we propose the second model that does not use Web crawlers and offline processes. Figure 3 illustrates our proposed architecture model. The user interface allows the interaction between users and system, accepts user's input and present results to the users. This tool has two main parts. Part 1 is independent and includes a code search component; and part 2 includes processes for finding related Web documents for each code sample. Input of the part 2 is each code sample that is obtained from search results of the part 1.

With respect to the part 1, it is simple to implement an efficient Code Search component by using Code Search service API. This component gets the user's needs to build a request, send it to the service, and retrieve response results that are ranked by the Code Search API's algorithm. Then, we can customize and format these results before presenting them to the user interface. Details of the processes in the part 2 were described at Sect.3.

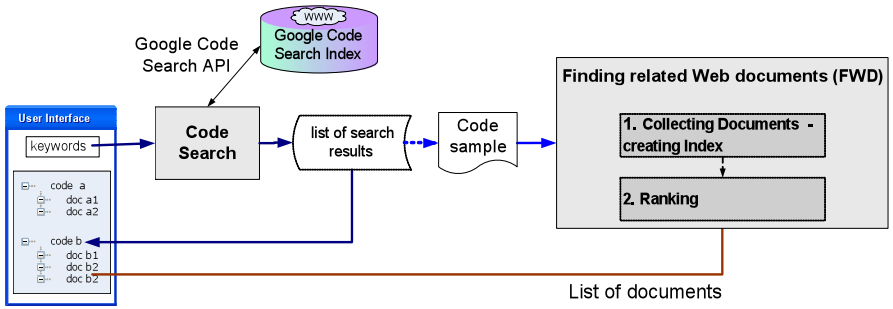

Figure 3: Experimental architecture model using FWD approach

5. Implementation

We implemented a Web-based demo application. This section describes the implementation, also the issues, challenges and solutions during the designing and developing this system.

Implementation is an application written by Java programming language (Java Servlets, JSPs, JavaScript). Java Beans are for presenting and storing the search results. Servlets control processes in the application interacting with Web server. Algorithms and manipulating processes are implemented by Java classes. User interfaces are implemented by JSPs and JavaScript. Figure 4 demonstrates all main components in the implementation.

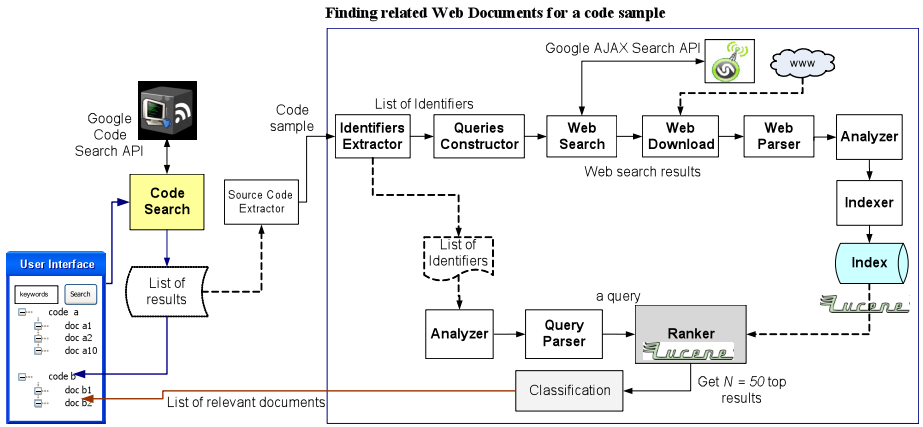

Figure 4: Main components in the Web-based demo application

5.1 Code Search

For implementing Code Search component, we utilized Google Code Search Data API client library. This API allows developers to create a plug-in for IDE or client, web applications to access code repositories indexed by Google Code Search. To send a query to Google Code Search, we can use regular expressions in queries. The result returned for a request is a feed containing entries. Every entry represents for a search result. The elements in an entry in a code search feed include: information about the authors, license applies to the code; the full path to the file, name of the file in the version-control repository or archive; the matched code snippets; the location of the package (name and URI). In case of using the client library these elements are represented by a set of classes and parsed into appropriate objects.

5.2 Web Search

Here, Google code search API has not supported API to get the content of source file in its result entries. We utilized HtmlUnit library[b] to implement a module named Source Code Extractor to automatically parse Google's result page and extract the content of code.

Next, Identifiers Extractor component is responsible for parsing a Java source file and extracting identifiers. We implemented a class that extends the class *ASTVisitor* from the Java DOM/AST library of the Eclipse JDT. To optimize the later downloading process, some common import declarations (for examples, *java.util, java.io, java.text, etc*) are eliminated from the identifiers' list. Using words from list of identifiers, Queries Constructor component will create queries.

For obtaining information of Web documents, we utilized Google AJAX Search API to call Google Web search service for these queries. In fact that, after some preliminary experiments comparing performance between the Google and Yahoo! Web search API, we realized that Google AJAX Search API allows retrieving results faster. Currently, we only retrieve around the first ten result pages to the fifteen result pages for each request. In our implementation, we found that looking at more pages did not significantly improve the quality of the results and caused to cost long time for downloading. Because Lucene always digests text data, before these web pages are indexed, they need to be parsed to extract text content. We implemented a Web parser that uses open source Apache Tika toolkit. We also implemented an analyzer for transforming all capital letters into lower case letters, removing common words and stemming. The common words include some English stop-words and Java stop-words.

b) http://htmlunit.sourceforge.net/

5.3 Indexing and Ranking with Lucene

In our implementation, we used the extended VSM scoring that refined by Lucene[6] with some additional factor to find the best matches to queries. Lucene called this formula *Lucene's practical scoring function* as follows.

$$\text{score(q,d)} = coord(q,d) * queryNorm(q) * \sum_{t\ in\ q} (tf(t\ in\ d) * idf(t)^2 * t.getBoost() * norm(t,d)$$

tf(t in d) is a factor correlates to the term's frequency; *idf(t)* is a factor correlates to the inverse of the term's document frequency; *coord(q,d)* is a score factor based on how many of the query terms are found in the specified document; *queryNorm(q)* is a normalizing factor used to make scores between queries comparable; *t.getBoost()* is Boost of term t in the query q as set by application (calls to *setBoost()*); *norm(t,d)* is s value encapsulates a few (indexing time) boost and length factors.

The Indexer component is implemented by using Lucene. Lucene provides some core classes for indexing text such as *IndexWriter*, *Directory*, *Analyzer*, *Document*, *and Field*. The *IndexWriter* class is main point of the indexing process. It creates a new index or opens an existing one, and then adds, removes or updates *Documents* in the index. The abstract class *Directory* represents the location of a Lucene index. *Document* and *Fields* are two Lucene's fundamental units of indexing. Every *Document* contains some named fields that are embodied in a class called *Field*. Each Field has a name and a series of detailed options that describes what Lucene should do with the Field's value when the document is added to the index. In our implementation, the most important Field is "Contents" that is combined from title, summary and text content. To distinguish the different important levels of Documents, higher Boost values are set for some Documents. If a page has its domain appearing in our predefined list of domains, its Boost value is greater than default value (1.0f). We defined some important domains such as *sun.com*, *java.net*, *developer.com*, *ibm.com*, *oreilly.com*, *javaworld.com*, *java2s.com*, *sourceforge.net*, etc.

To support building query and basic search operations, Lucene provide some necessary classes such as *IndexSearcher*, *Term*, *Query*, *TermQuery, TopDocs*, *ScoreDoc*. *IndexSearcher* is a commonly used subclass that allows searching indices stored in a given directory. Its *search* method returns an ordered collection of documents ranked by computed scores. We can decide the number of top search results that need to be retrieved by specifying it in the *IndexSearcher*'s search method. Searching for a specified word or phrase involves wrapping them in a term, adding the terms to a *Query* object, and passing this query object to *IndexSearcher*'s *search* method.

Lucene also provides various types of concrete query implementations, such as *TermQuery*, *BooleanQuery*, *PhraseQuery*, *PrefixQuery*, *SpanQuery*, etc. We can construct powerful queries by combining any number of query objects using *BooleanQuery*. Our Query Parser component that is responsible for constructing a query from the list of identifiers is implemented similar to this method. Additionally, every term query has different contribution to the total query. The term queries that their text terms extracted from type declarations such as major class, interface names are set a higher boost value than from super class, super interfaces, import declarations. These terms are usually common with programmers then their contributions to score of documents are not high (term queries are set lower Boost values). Some of them are *byte*, *short*, *int*, *float*, *list*, *void*, *java*, *javax*, *util*, *io*, *text*, etc. Specially, through validating some experiment results, we explore that including all of terms of import declarations into the query can cause a problem that some unimportant documents are ranked into top of results. This problem is solved by only extracting the last term of each import declaration when building a query for the source code.

ScoreDoc and *TopDocs* are primary classes involved in retrieving the search results. *ScoreDoc* is as a simple pointer to a document contained in the search results. This encapsulates the position of a document in the index and the score computed by Lucene. The *TopDocs* class is a simple container of pointers to the top N ranked documents that match a given query. In our implementation, to score the text documents, we implemented the Ranker module, that its input is a query constructed from the list of code identifiers and returns N = 50 results.

5.4 Classification

When presenting the results to the user interface, the pages that are possibly Javadoc will be attached with specific icons. In order to do that, we implemented a simple solution through observing Javadoc pages on Web. We realized that their header or footer usually contains some of the following keywords: "Overview", "Package", "Class", "Use", "Tree", "Deprecated", "Index", etc. Searching in the results collection for documents matching with a query constructed from these words, we can recognize Javadoc pages with a relative accuracy.

In addition, we used the Highlighter package, which was also released together with the full Lucene's source to fragment and highlight text parts in the results based on the query. So, matched parts in the results will be highlighted, once being presented to the user interface.

6. Case Studies

We took some explanatory case studies validate the quality of results for our proposal automatically finding Web documents for code. To measure the retrieval effectiveness of an IR system, two most widely used measurements are precision and recall. Recall is the ratio of correct documents retrieved for a given query over the number of relevant documents for that

query in the set of documents. Precision is the ratio of the number of correct documents retrieved over the total number of documents retrieved. However, in our case, calculation of recall for each request is very difficult or impossible, because of the following reasons: (1) collection of Web pages always changes, and there have been a huge number of pages, and (2) validating whether a result is correct (useful) is to depend on satisfaction level of users and their knowledge.

We assume that, searchers are usually interest in some results in the top ranked results. So that, we defined a new measurement called *useful rate* (similar to *precision*) that is the ratio of the number of useful links over the number of total links retrieved and is measured at certain cut-off levels. Experimental results are shown in Table 1. For these results, whether a page is assumed to be useful was based on inspection and feed back of three participants. Although, we can have knowledge about the related concepts, these results can be different by validation of different groups according to their satisfaction levels. There are several conventions in our tests as follows: the code samples selected in the experiment are the source files that use external APIs and include many identifiers. Some pages that duplicate content but have different addresses can be counted. Besides, we gave some following definitions. Give a code sample A that can include (relevant with) some new concepts such as $a_1, a_2..., a_n$. The new concepts can be new types (code identifiers) that refer to other packages (APIs), code ideas, etc. Document B is assumed to be useful (or relevant) for comprehending A if B satisfies one of the following conditions: B is a page describing A such as Javadoc page, or explaining usage of A as tutorial page; B talks about element a_i; B contains an issue or solution similar with what A is solving; B describes some domain concepts that the users need.

The test cases were taken on a PC that has the following configuration, Intel Core 2 Duo CPU, 2.40 GHz; RAM 4.00 GB, connected to ADSL Internet.

The results showed that the lower cut point is, the higher useful rate is. Useful rate is relatively high around 60~90%. This means our approach is efficient in ranking documents; we can get many useful web documents at the top of results. During the experiments, we attempted to classify the results manually and realized that kinds of useful documents that our tool can found are: web pages containing similar code snippets, Javadoc pages, web pages containing explanatory text of usage APIs, tutorial pages, pages from mail listing or groups, and forums that discuss about related issues according the code sample, etc. These kinds of above information and feed back from participants showed that our approach is useful for comprehending an existing program. In other words, automatically finding related Web documents for search results of code search engines are good for searchers to get new domain knowledge in code sample.

This experiment also shows that the implementation needs much time for downloading and processing, about from 10 seconds to 5 minutes depending on the number of declared names in a code sample. An infrastructure with parallel processes can improve the system performance. Using the enterprise architecture model as described in Sect.4 could be a better solution.

Code Search			**Results of finding related Web documents for each code sample**				
Keywords to search	**Code sample selected (.java)**	Ranked order	**Cut point**	**Useful links retrieved**	**Irrelevant links retrieved**	**Total links retrieved**	**Useful Rate (%)**
"HtmlUnit"	**HtmlUnitTestCase**	2	1	13	2	15	86.7
			2	25	5	30	83.3
			3	34	11	45	75.6
"indexHTML"	**indexHTML**	1	1	14	1	15	93.3
			2	27	3	30	90
			3	39	6	45	86.7
"XML"	**XML**	1	1	13	2	15	86.7
			2	25	5	30	83.3
			3	36	9	45	80
	NodeTree	9	1	12	3	15	80
			2	19	11	30	63.3
			3	25	20	45	55.6
"ASTParser"	**ASTParser**	1	1	11	4	15	73.3
			2	17	13	30	56.7
			3	21	24	45	46.7
"HTML Parser"	**HTML**	13	1	12	3	15	80
			2	18	12	30	60
			3	21	24	45	46.7
	DocumentParser	15	1	14	1	15	93.3
			2	25	5	30	83.3
			3	35	10	45	77.8

Table 1: Results from several case studies

7. Related work

This section discusses several related studies: 1) program comprehension, 2) traceability links between code and documents in software engineering, 3) assumptions when proposing the method, and 4) types of information programmers usually need during developing software programs.

In the area of assisting for program comprehension, academic research projects have primarily focused on extracting properties from the program code or code comments or combination both of them [9]. Their goals are to syntactically analyze the source code or information encoded in comments and identifiers for creating event concepts and relationships in the form of a concept-based semantic network. Search based software engineering with search-based optimization algorithms have been also applied to program-comprehension related activities [10].

Recovering traceability between code and specifications [11, 12, 13] is also useful for program comprehension. The experiment results showed that Vector Space IR performs as well as Probabilistic IR. Vector Space Model requires less effort in the preparation of the query and document representations [37]. Recently, a comparison [8] showed that, for realistic datasets, VSM implemented by Lucene is one of the best models that do not perform dimensionality reduction. However, while the above researches are limited in local repository where stores source code's software and its manuals; we applied the proposal for finding documentations on Internet.

One more similar point between our research and the previous researches is that we assumed that programmers use meaningful names for program's items, the source program and its documents are often likely to share common terms.

Finally, the researchers [14] showed that many searches by programmers are interested the most in information about APIs. Our approach used declared names in code sample as queries to retrieve Web pages. These factors can contribute to the result pages tending to focus on API documentations, seek more information about a particular API.

8. Conclusions

In this paper, we discussed some barriers when programmers search open source by using the current source code search engines. We introduced a method named FWD that allows automatically finding the related documentation on Web for code sample. We discussed usage of the method by two models of code search engine that can recommend related Web documents for its results. To realize this approach, we implemented a Web-based search tool for assessing. The paper showed some preliminary case studies to evaluate the quality of the approach FWD. The results demonstrated that our proposal can find useful web documents' links that tend to appear in the top results and they are useful for searcher in program comprehension.

However, more case studies with code samples from other resources for validating the tool should be done in the future. The experiments need inspection of other experts. Besides, some enhancements that we would like to adopt to our approach in the near future can be summarized as follows: (1) study other optimizing solutions to improve the implementation's performance; (2) investigate alternative methods for query construction (identifiers combination); (3) automatically recognize and classify types of web pages such as code snippets, tutorial pages, forums/blogs, etc in the result pages.

References

[1] Google Code Search's Website, <http://www.google.com/codesearch/>.

[2] Koders' Website, <http://www.koders.com/>.

[3] Krugle's Website, <http://www.krugle.com/>.

[4] Sourcerer services' Website, <http://sourcerer.ics.uci.edu/services/>.

[5] The SPARS Project, Osaka University, <http://demo.spars.info/j/>.

[6] Apache Lucene Scoring, Apache Lucene, viewed 28 Nov. 2009, <http://lucene.apache.org/java/3_0_0/scoring.html>.

[7] Aharon A., Mordechai N., and Yahalomit S., 'A Traceability Technique for Specifications', In *Proc. of the 16th IEEE International Conf. on Program Comprehension*, IEEE, pp. 103-112, 2008.

[8] Antoniol G., Canfora G., Casazza G. and De Lucia A., 'Information Retrieval Models for Recovering Traceability Links between Code and Documentation'. In *Proc. of the IEEE International Conf. on Software Maintenance (ICSM)*, San Jose (October 2000)

[9] Bradley L. Vinz and Letha H. Etzkorn. A Synergistic Approach to Program Comprehension. Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, 2006.

[10] Mark Harman. Search Based Software Engineering for Program Comprehension. 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE, 2007.

[11] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", IEEE Transactions on Software Engineering, vol. 28, no. 10, October 2002, pp. 970 – 983.

[12] Marcus, A., Maletic, JI, and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", Int. Journal of Software Engineering and Knowledge Engineering, vol. 15, no. 4, Oct. 2005, pp. 811-836.

[13] Xiaobo Wang, Guanhui Lai, Chao Liu. Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering. Journal Electronic Notes in Theoretical Computer Science 243, 2009, pp. 121–137.

[14] Raphael H., James F., Daniel S. Weld, 'Assieme: Finding and leveraging implicit references in a web search interface for programmers', In *Proc. of the ACM Symposium on User Interface Software and Technology*, October 7-10, 2007, USA, pp. 13-22, 2007.

[15] Google AJAX Search API, <http://code.google.com/apis/ajaxsearch/web.html>.

[16] Yahoo! Web Search APIs, <http://developer.yahoo.com/search/web/>.

[17] Codase search engine's Website, <http://www.codase.com/>.

[18] O. Hummel, W. Janjic, and C. Atkinson, 'Code conjurer: Pulling reusable software out of thin air', *IEEE Software.*, 25(5): pp 45–52, 2008.

[19] S. Thummalapenta and T. Xie, 'Parseweb: a programmer assistant for reusing open source code on the web', In *ASE '07: Proceedings of the 22th IEEE/ACM international conference on Automated software engineering*, New York, pp 204–213, 2007.