Indexing the Java API Using Source Code

Homan Ma, Robert Amor, Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
{trebor,ewan}@cs.auckland.ac.nz

Abstract

The basic idea behind software reuse is that software developers use reusable components *found in* software repositories *to reduce the amount of code that has to be written and so increase productivity. A problem arises, however, if the repository is too big — it becomes difficult to find relevant components. What is needed is an effective means to query repositories. Most approaches to developing such means involves creating a good* index *to which the queries can be applied. Developing a good index requires identifying the relevant information on which to base the index. In this paper, we present the results of a project that used* source code *as the basis for the index.*

Keywords: Java Standard API, Software Repositories, Source Code Analysis, Reuse

1. Introduction

The basic idea behind software reuse is that software developers use *reusable components* found in *software repositories* to reduce the amount of code that has to be written and so increase productivity [17]. The success of a repository depends on the number of components it contains. If a repository has too few components, then a developer is unlikely to get much benefit from it. However repositories with many components face a different problem — how a developer finds the particular component she needs, or, failing to find something relevant, how to be sure that this is because there is nothing relevant in the repository. What is needed is an effective means to query repositories, ideally using *free-text* queries. Most approaches to developing such means involves creating a good *index* to which the queries can be applied. Developing a good index requires identifying the relevant information on which to base the index. In this paper, we present the results of a project that used *source code* as the basis for the index.

The rationale for our approach is as follows. Anyone using a repository would have done so to solve a problem in a specific domain. If they follow good programming practise, they would name the identifiers in their code in a way that reflects their use. We would hope then that the identifiers relating to the use of entities from the repository would reflect the given domain's view of those entities. Given uses of a repository entity from multiple domains, we would build up a vocabulary of how programmers think of that entity upon which we could build an index.

There are two main problems with this approach. The first is, we need to find "enough uses" of a repository. For this, we have a Software Corpus consisting of a number of open-source Java applications [22]. As most Java applications use the Standard Java API [26], we would hope that we would have enough uses of it to allow us to create an effective index of the Standard API. The second problem is, there is an important assumption in our rationale — that programmers really do use meaningful names. Whether or not this is the case is something we need to establish.

In this paper we present our study into the use of Java source code to create an effective index to the Java Standard API. The rest of this paper is organised as follows. In the next section, we discuss related work and introduce the concepts needed to explain our work. In section 3 we discuss how what choices we considered for the various ways we could use identifier information to create and index, carry out queries on it, and evaluate the results. Section 4 presents our results and these are then discussed in section 5. Finally, we give our conclusions in section 6.

2. Background and Related Work

The problem we are addressing is an instance of the problem identifying documents from a set of documents that are most relevant to a given free-text query, which is the domain of Information Retrieval (IR). The main issues we feel we have to address are synonymy and polysemy. Synonymy is that there are different words with a similar meaning. For example, "hot" might be considered synonymous with “spicy”, “warm”, “eager” and “attractive”. Polysemy refers to the fact that one word can have multiple meanings. For example, the word “plant” can mean vegetation or machinery. The use of words to describe the desired component is based on different contexts, knowledge, needs, or linguistic habits of programmers. Programmers can therefore describe the same component differently. Furnas et al. measured the probability of two people choosing the same keyword for a single well-known object as less than 20% [4]. The IR community has explored many avenues for dealing with this problem, and others have tried applying IR techniques to the problem of indexing software repositories.

One such attempt is by Maarek et al., who investigated the use of attributes extracted from natural language documentation by using an indexing scheme based on the notion of lexical affinities [15]. What is particularly of interest to us is their motivation for indexing from documentation rather than source code. They concluded, on examination of samples of code that “even when dealing with well-written code, there is a very low probability that the programming styles of the various pieces of code will be consistent” and that “a single programmer may use totally different identifiers for expressing the same concept.” We disagree — rather than being a disadvantage this may actually be an advantage as it potentially provides more information about a concept. However, exploiting this property requires some means to correlate different words used for the same concept.

The basis for many IR techniques is the Vector Space Model (VSM). In this approach, every document and also the query is represented as a vector with an entry representing the occurrences of a given term [24]. All terms are sorted and their positions in the sorted set are used as indices in the vectors. Most of the vector entries will be zero for most documents, since the distinct terms found within a document are only a subset of the terms found in the collection of documents. This means that a sparse matrix representation is often appropriate. The *document vector matrix* (DVM) is then the matrix where each row represents a given term, and each column represents a document.

One of the advantage of using the DVM is the fact that the similarity between different terms, between a term and a document, and also between different documents in the collection can easily be calculated. However polysemy and synonymy will reduce the effectiveness.

In the last few years there have been various attempts to use *latent semantic analysis* (LSA) or *latent semantic indexing* (LSI) [1] to index the Java API JavaDoc pages. This technique uses uses Singular Value Decomposition (SVD) [5] to reduce the DVM space, essentially merging dimensions. Since each dimension corresponds to a term in a document, the effect is as if the higher level concepts corresponding to the merged dimensions have been identified. The first use was by Ye [28, 29]. He developed CodeBroker, which provides retrieval of *methods* from the Java API. It uses LSA to provide free-text indexing, and augments the retrieval process by using *signature matching* to extract the more suitable components identified by the LSA process. CodeBroker is developed for Emacs.

We also have previously explored using LSI on the documentation of the Java API, in particular the JavaDoc pages [11, 12]. As well as trying to reproduce Ye’s results, albeit in a different development environment (Eclipse), we also looked at whether document granularity had an effect. We compared the results treating each JavaDoc page as a separate document with treating each method section of a JavaDoc page as a separate document. While there were many more method documents, the thought was that the class documents would provide more context. We also compared use of free-text queries with LSI based indexing with straight keyword search and Google. Although it was difficult to compare our results with Ye’s, our results appeared similar. Our results also indicated that the class documents were more effective for use with LSI than method documents, and LSI often performed better than Google (with Google searches sometimes not returning any result). As we use this previous study to evaluate our work, we will give more details of it later.

Similar work was carried out by Jensen [9], also based on Ye’s work. Jensen obtained similar results to ours with respect to the quality of retrieval, however unlike our work, his system was not integrated with an IDE.

We are aware of no other work that specifically uses identifiers to index software repositories, and in fact there is only a small amount involving identifiers at all. One example is by Michail and Notkins, who considered finding components that are similar across libraries [18]. They argued that one way to do so was by name matching, as names given to different libraries are meaningful and therefore a component with the same name in different libraries is likely to serve a similar purpose. One relevant aspect of their work to ours is how they treated identifiers. They observed that many identifiers are really several words concatenated together and these words needed to be identified. They did so by using various heuristics for detecting word boundaries, such as use of underscores or change in case. They stated that although not a particularly good style of programming, some developers may not separate words at all and therefore investigated the use of a dictionary lookup to break up words further.

Another project, by Zaremski and Wing investigated the use of signature matching in organising, navigating through, and retrieval from, software libraries [30]. This uses type information from method signatures, possibly extracted from code. In contrast with Maarek et al., Zaremski and Wing argue code is a more reliable source of information than program specifications. They state that, “we cannot, as yet expect programmers to document their program components with formal specifications”.

Also relevant to this paper is a study completed earlier in the project investigating the usage patterns of the Java Standard API [13]. We needed this information in order to determine how much information we would have available to us for the indexing. We studied 76 open source Java applications from our corpus looking at the use of version 1.4 of the API. We found the usage to be much lower than anticipated. About 50% of the Java classes and about 80% of the Java methods are not used at all. While such low usage would restrict what we could index, there seemed to be a lot of commonality between applications, suggesting that we might at least be able to effectively index the common subset.

WordNet is a manually-constructed online lexical system developed by George Miller at the Cognitive Science Laboratory at Princeton University [19]. Originally, the goal of the WordNet project was to produce a dictionary that could be searched conceptually instead of only alphabetically, WordNet has evolved into a dictionary based on current psycholinguistic theories of human lexical memory.

We believe that WordNet can be used to help deal with the problems of polysemy and synonymy. Documents related to the query sometimes contain only the synonyms of the query words instead of the query words themselves. We can use WordNet to create new versions of a query by replacing words in the original query with related words.

3. Methodology

Our basic hypothesis is that we can create an effective index for the Java API using information found within variable identifiers found in open-source Java applications. We have developed a system to test this hypothesis. It consists of several phases. The first phase extracts the identifier information from the source code from our corpus. The next phase is the data preparation phase, which involves the organisation of the variable identifier data. The third phase is the indexing phase, in which an index of the Java API using the variable identifiers is created. The last phase is the query phase, in which queries are submitted to the system. The extraction phase is done effectively once. The data preparation and indexing phases are done several times varying different options as described below. The query phase is then applied as often as needed on the index resulting from each indexing phase.

3.1 Corpus

As mentioned earlier, our study uses a corpus of Java software we have been developing [22] and was also the subject of our earlier study on API usage [13]. Due to the way we have developed our corpus, we organised the applications into two sets: “Set 1” was the original set of 39 applications that were in the corpus when this project began, and “Set 2” was the 37 we added with the intent to improve the representativeness (see [13] for their membership). While the two sets did not show much difference in the API usage patterns, we maintained this distinction for this study. This allowed us to investigate to what degree the choice of data source could affect our results. We also had the implementation of the Java API itself as a source of identifier information.

The reason we considered these different groups was due to performance reasons. Considering all applications together required substantial time and memory as the larger the set of applications, the more terms, and so the larger the DVM. We were interested in determining whether or not we could get acceptable effectiveness with a smaller set.

3.2 Identifier Information Extraction Phase

In the work reported here we are interested in the identifiers associated with the uses of the API. One attribute of interest is the length of identifiers. Variable identifier lengths can indicate amount of information that the variable identifier holds. For example if we found two variable identifiers for the *String* class, *reservedWordStart* and *s*, then *reservedWordStart* holds more information about what the *String* object is being used for than the identifier *s*.

What we found was that in the corpus we analysed the most common identifier length was 1, with the next most frequent length being 4. As we were unsure of the consequences of including many short identifiers, we decided to classify applications as “good” or “bad” depending on whether or not an application had a identifier length greater than 6 characters with a frequency greater than 5% of all identifiers used within the application. This added another data source to those identified above.

As we had discovered with the API usage results, the API implementation itself showed quite different characteristics than the other applications. When considering the applications, we found 1,288,711 unique variable identifiers associated with the use of the API, whereas there were 2,039,735 used in the API implementation (since all identifiers in the API implementation had to be associated with the use of the API). Furthermore, the most common identifier lengths were mostly in the range 10 – 15 (there were only 153 identifiers of length 1 for example). This might suggest that the developers of the Java API took more care in their choice of identifiers.

Having the identifiers was not sufficient by themselves. Many identifiers are actually multiple words concatenated together (such as *reservedWordStart*) and so we needed to extract the individual words.

In order to separate identifiers into multiple terms, two forms of identifier splitting were employed. The first approach was by splitting the identifiers according to the Java convention. The Java code convention is that multiple word identifiers should be joined by camelcase, so the identifier *aMultipleWordIdentifier* would become *a multiple word identifier*. Acronyms were also identified in this process for instance *checkURLvalidator* would become *check URL validator*.This handling of identifiers is similar to the approach by Michail and Notkin [18]. Michail and Notkin have used abbreviation expansion and dictionary-lookup, which is also implemented in our system.

The second approach was by splitting the identifiers using the lexical online dictionary, WordNet [27]. Parts of identifiers were run through WordNet to see if they were actual words. This is because the Java convention approach above assumed that all developers of the open-source applications within the software corpus followed the Java Convention. However, while it may not be good practice, some developers may not follow the convention. This means that, for example, *isenabled* would not be correctly split into *is enabled*. Sometimes, it is also not always clear whether a concept is written as one or two words, such as with *Scrollbar* versus *ScrollBar*. By using a dictionary we compared substrings with the words in the dictionary to see if they are actual English words or not. To avoid ambiguity, longer words are preferred, as having more specific words would be a better distinguishing factor.

Some words were not recognised by WordNet as actual words but were in fact actual words used in the software domain such as *scrollbar*, *checkbox*, *textfield*. We generated a list of software domain words not present within WordNet and use it together with WordNet during the comparison process. This means when a word is compared to WordNet to check if it is an actual word, it is also checked with the software domain word list, if the word is found to be present within either WordNet or the software domain word list, then it would be considered an actual word.

On closer observation of application source code it was clear that the use of abbreviations within variable identifiers was very common. Manual studies showed the presence of software specification abbreviations such as *s*, *url*, *ex*, *src*, *btn* and single characters. These abbreviations caused problems when matching abbreviated words in documents to the query. To avoid these problems, a table of common abbreviations was manually generated through domain analysis. No existing domain specific abbreviations table could be found relating to software. Those that exist are only for some well-known domains.

Through the manual process of running through the list of words and finding the associated expansion, a 382 word software specific abbreviation lookup table was generated. There was some ambiguity however when expanding some abbreviations that have more than one expansion. For example, *dns* could be expanded to *domain naming server* or *dynamic naming system*. To choose the right expansion at runtime would require an understanding of the context in which the abbreviation is used, however for simplicity for our system the expansion that is most often used with the abbreviation was chosen.

3.3 Data Preparation Phrase

In this phase we organise the variable identifier information extracted in the first phase into documents according to the Java hierarchical structure. That is, an identifier used in relation to a particular class in the API will be divided into words as described above and added to a "document" corresponding to the class. For example, an occurrence of *ArrayList nameList* in the source code will result in the phrase *name list* being added to the text document *java/util/ArrayList.txt*.

3.4 Indexing Phase

Using the data formatted in the data preparation phase, a DVM is then created. The indexing phase is subdivided further into four steps. The first step is to do a frequency cut-off where terms appearing outside a given frequency range are ignored. In other words, terms occurring less than *lowerLimit* number of times within a document would be removed. For example given a document (*java.util.LinkedList*) containing the terms *name list name list object*. Applying a limiting function with a lowerLimit=2, where terms appearing less than 2 time would be removed, would mean only the terms *name* and *list* would be kept and *object* would be ignored. Jones [10] states that terms appearing less often in a collection of documents are better at distinguishing documents from each other. Less frequent terms within a document also cause unwanted noise. By reducing the number of terms in the collection through this step, the complexity of the DVM is reduced, and as an advantage would mean better retrieval performance.

The second step is the removal of stop-words. Stop words are words that are generally regarded as too common to be good at discerning documents. Jenson surmised that elimination of stop words from his system could potentially give better performance, however this is untested in his system [9]. Our stop word list included words related to software such as *string*, *int*, *boolean* and most of the contents of the *java.lang* package.

The third step is the application of stem removal algorithms on the terms in the documents. Stem removal is a method used to remove word inflections in order to reduce the number of different terms for the same base word [21]. Terms with a common stem will usually have similar meanings. Simple stemming usually involves the removal of a word's suffix. For example words like *connect*, *connected*, *connecting*, *connection* and *connections*. All these terms have a similar meaning and therefore can be reduced into to a single term *connect*. The affect of stem removal is that the number of terms found within the DVM matrix would be reduced, as like terms will be reduced to a single term through the process of stem removal and as a result we would expect some performance gain. Various suffixes -ED, -ING, -ION, -IONS are also reduced. Three different Algorithmic stemmers were used in the development and testing of the system — Lovins, Porter, and Snowball. Snowball is an extension of the Porter stemmer.

Stem removal generally improves recall but leads to a loss of precision. There are a lot of variations in the results obtained from using stemming. Some have found the improvements in recall to be more significant than the losses in precision [6]. Others have had opposite results [2]. However, in the detailed analysis conducted by Hull [8], he has concluded that stemming in general is almost always beneficial.

The final step in the creation of the DVM is by selecting the "term weighting" method. This determines what actual values occur in the matrix. While using just the frequency of occurrence of terms in a document is simplest, it may not always be best. Manning and Schütze [16] observe that whether or not a term appears in a document at all should have more significance than how many times it appears. This suggests that some kind of weighting of terms should be used in the DVM. According to Dumais [3], usable weight schemes include *binary occurrences*, *tf-idf*, and *term frequency*.

Term frequency weighting gives a local weight that is exactly the frequency of the term. As already observed, this scheme possibly mis-represents the importance of the introduction of a new term. Binary weighting assigns 1 if the term is present (no matter how many times) and 0 if it is not. Binary weighting as it is is too simple as it retains too little information, and it is rarely used in practise.

It is a common assumption in IR that if a word occurs in almost all documents, it is less suitable to distinguish documents and therefore it should have a less weight. The *tf-idf* weight (term frequency-inverse document frequency) is often used in information retrieval and text mining [7] to evaluate how important a term is in a document. The importance of a document increases proportionally to the number of times a term appears in the document but is offset by the frequency of the term in the corpus.

3.5 Final Phase: Querying

Queries are provided as strings, such as *judge if a file is a directory*. The query is taken in as a string and is then split using the same process described in the data preparation phase. This means that identifier splitting is performed and numbers are removed. The query is then treated as a (pseudo) document and its VSM representation can then be determined.

To deal with the problems caused by synonymy and polysemy queries are processed using WordNet. Two different methods were employed. The first method was for each word in the query to be expanded to all synonyms possible. For example, the query *append two strings* would be expanded to *add append supply add on tag on hang on two deuce string chain strand thread*. By doing so, it would give all possible terms relating to the query and hence it was hoped would return the components with terms that were only synonyms of the words found within the query, whereas before they were not returned as they did not have terms matching the query. However this gave a very poor result.

The second method we tried was to change each word of the query one at a time to form a set of queries that were then run through the system. The average rating for the components returned was then calculated and used to represent the relevancy of the returned components. The affect of doing so was also so that components with terms that were only synonyms of the words found within the query would be returned, whereas before they were not returned because they did not have any terms matching the query. For example the query above would form the set of queries: *append two strings*, *add two strings*, *supply two strings*, *add on two strings*, *tag on two strings*, *hang on two strings*, *append deuce strings*, *add deuce strings*, *supply deuce strings* and so on. What is interesting about the second method is how to combine the results of the queries given. Do we select the set of components that are present in the retrieved components of all queries? How do we calculate the rating of the returned components? Do we ignore queries that have fewer or more than *n* number of retrieved components? Previous work using WordNet has not encountered this problem and a more detailed study into the development of a proven algorithm to handle the aggregation of the results needs to be developed in future studies. For this study the system returned the result that appeared in the most queries.

In order to rank the results, all documents are compared to the query pseudo-document by computing the angle between the VSM representations of each document and the representation for the query pseudo-document. The smaller

Query ID	Free Text Query (**F**)	Keyword Query (**K**)	Desire Components (Java Class)
1	judge if a file is a directory	file directory	java.io.File
2a	split a string into substrings	string split	java.lang.String
2b		substring	java.lang.String
3	determine if it is a leap year	leap year	Java.util.Gregorian-Calendar
4	return true if the string is capitalized	capitalized String	java.lang.Character
5	change the file name	change file name	java.io.File
6	append two strings	append string	java.lang.String java.lang.StringBuffer
7	create a directory on a floppy disk	create directory	java.io.File
8	create a random number between two numbers	random number	java.lang.Math java.util.Random
9a	check if a character is a digit	character digit	java.lang.Character
9b		digit	java.lang.Character
10	a listener for mouse movement	listener mouse movement	java.awt.event.Mouse-WheelListener
11	draw a circle fill with red color	color circle	java.awt.Graphics java.awt.Color
12	establish a connection between two computers over internet	connection internet	java.net.Socket
13	round a double to integer	round integer	java.lang.Math
14	sort a long array into descending order	sort array descending	java.util.Arrays
15a	create a application that runs on browser	application runs on browser	java.applet.Applet
15b		application browser	java.applet.Applet

Table 1. Queries used

the angle, the higher the perceived similarity. The resulting list is sorted by decreasing similarities. The top *n* (a configurable value) documents with a non-zero similarity value are returned along with the number of documents retrieved.

3.6 Implementation

The system was written in Java. It uses JWordNet [25] to interface with WordNet. The vector representation of documents was managed using The Word Vector Tool is an open-source Java library for handling statistical modelling [24]. To handle the complex data formats represented in IR systems, and in particular to compute the similarity values between documents and queries, we used the Network Common Data Form (NetCDF) package. It consists of an interface for array-oriented data access and a library that provides an implementation of the interface [23].

3.7 Experimental Setup

We use the same queries we have used in our earlier work so that we could use our earlier study as a benchmark [11, 12]. Following our earlier work, we considered two kinds of search query — free-text queries, which were the main target of our research, and keyword queries. Keyword queries were considered because experienced programmers might use these as short-cuts. They also allowed for a fairer comparison with Google.

We used 15 free-text queries, which included 8 from Ye's work [28], which are shown in table 1. The queries were chosen to be representative of the kind of free-text query a developer might make in the midst of writing code, and no attempt was made to "sanitise" them. From the free-text queries we created 18 keyword queries, with 3 free-text queries generating 2 queries each (IDs 2, 9, and 15). In the results presented later, the free-text queries will be identified with the ID given in this table pre-pended by **F** and the keyword queries pre-pended by **K**. Some of the queries have multiple possible desired components. These are treated as

Query	Set 1	Set 2	API	Good	All
F1	**1**	**1**	**1**	**1**	**1**
F2	**17**	**18**	**18**	**18**	**20**
F3	-1	-1	-1	-1	-1
F4	-1	-1	-1	-1	-1
F5	**8**	**8**	**8**	**8**	**10**
F6a	**14**	**17**	**16**	**13**	**17**
F6b	**9**	**8**	**7**	**12**	**9**
F7	**13**	**12**	**9**	**13**	**14**
F8a	-1	-1	-1	-1	-1
F8b	**9**	**8**	**7**	**12**	**9**
F9	**1**	**4**	**10**	**1**	**4**
F10	**2**	**2**	**2**	**1**	**1**
F11a	-1	-1	-1	-1	-1
F11b	**2**	**3**	**2**	**2**	**2**
F12	84	-1	-1	77	107
F13	-1	-1	-1	-1	-1
F14	-1	-1	-1	-1	-1
F15	**6**	**7**	**6**	-1	**10**

Table 2. Using different data sources for free-text queries.

Query	Set 1	Set 2	API	Good	All
K1	**1**	**1**	**1**	**1**	**1**
K2a	**17**	**18**	**18**	**18**	**20**
K2b	**1**	**1**	**1**	-1	**1**
K3	-1	-1	-1	-1	-1
K4	-1	-1	-1	-1	-1
K5	**8**	**17**	**8**	**8**	**10**
K6a	**14**	**21**	**16**	**13**	**17**
K6b	**19**	**23**	**23**	**21**	**24**
K7	**13**	**12**	**9**	**13**	**14**
K8a	-1	-1	-1	-1	-1
K8b	**2**	**5**	**2**	**2**	**2**
K9a	**1**	**3**	**4**	**1**	**1**
K9b	**1**	**3**	-1	-1	**1**
K10	**2**	**2**	**2**	**1**	**1**
K11a	-1	-1	-1	-1	-1
K11b	**1**	**1**	**1**	**1**	**1**
K12	84	-1	-1	77	106
K13	-1	-1	-1	-1	-1
K14	-1	-1	-1	-1	-1
K15	**3**	**4**	**4**	-1	**5**

Table 3. Using different data sources for keyword queries.

different variants of the queries when giving results.

4 Results

As indicated above there are a number of aspects that we could vary, specifically: the data source, the frequency cut-off, stemming algorithm, and term weighting. We tried most combinations. We only have room here for what we consider the most interesting. More details are available elsewhere [14].

4.1 Changing Data source

Tables 2 and 3 show the results for the varying data sources for the free-text and keyword queries respectively. The values shown for a particular data source and query refers to the position or "rank" within the list returned by the query that the expected component appears. A "-1" result indicates that the relevant component expected for the query was not returned. Rankings lower than 25 are highlighted. For example the query *F1* when applied to the index created from *Set 1* has value 1 indicating that the component expected for that query appear at the top of the list (closest match).

The results are that about one third of the queries produced no component at all, but if a component was returned, it was almost always in the top 25 of those returned, and in fact as often as not were in the top 10. There is no obvious data source that is superior to the others. In particular, the "good" applications showed no obvious difference from the others. A slight argument could be made that "Set 1" had the better results, and so for performance reasons we chose to use that in the later experiments.

4.2 Limiting term frequencies

Tables 4 and 5 show the results of dropping terms that occurred less often than a given lower limit. Only those queries for which we got results without limits are shown. The first column shows the results without a lower limit. With the exception of a very large limit (100), changing the lower limit has very little impact. There was some improvement in terms of performance as the lower limited increased. A lower limit of 5 is used for the rest of the results shown.

4.3 Stem Removal

Table 6 shows the results for the different stem removal techniques. In the interests of space we just show how many of each type of query that the expected component is returned in the top ten of all the components returned. For example, for Lovins stem removal algorithm, 4 of all the free text queries returned the expected component in the

	Frequency Limit						
	1	5	10	15	20	25	100
F1	1	1	1	1	1	1	17
F2	17	17	18	18	18	14	-1
F5	8	8	9	10	11	11	34
F6a	14	14	14	14	14	14	-1
F6b	19	20	20	20	20	21	-1
F7	13	13	15	15	15	15	-1
F8b	9	9	9	9	9	9	-1
F9	1	2	3	3	3	3	-1
F10	2	2	1	1	1	1	15
F11b	2	2	2	2	3	3	45
F12	84	81	81	81	81	81	38
F15	6	6	6	5	5	4	-1

Table 4. Set 1 limiting term frequencies (lower limit, free-text)

top ten of all components returned and 5 of all the keyword queries returned the expected component in the top ten of all components returned. There does not appear to be much difference due to the different techniques.

4.4 Term Weighting

Table 7 shows the results for the different term weighting techniques. Again, there does not appear to be much difference due to the different techniques.

4.5 Comparison with Lin's results

Table 8 shows the comparison with our previously reported results using JavaDoc pages and LSI (marked as "Lin"), including the results using Google. We have highlighted those results that we subjectively considered to be providing a useful answer. The cutoff value we have used is 25, that is, if no approach ranked the expected component less than 25 then we regarded that as no useful result was produced. We chose 25 because the results were either smaller than this, or noticeably larger than it, and it is a small enough number that it is reasonable that developers would look at a list that big. When multiple approaches produced a rank less than 25 we chose the one that was in our opinion most obviously the best, but allowed "ties." From these results, there is no obvious improvement using documentation over source code identifiers, or vice versa. Both showed some improvement over Google.

	Frequency Limit						
	1	5	10	15	20	25	100
K1	1	1	1	1	1	1	17
K2a	17	17	18	18	18	14	-1
K2b	1	-1	-1	-1	-1	-1	-1
K5	8	8	9	10	11	11	34
K6a	14	14	14	14	14	14	-1
K6b	19	20	20	20	20	21	-1
K7	13	13	15	15	15	15	-1
K8b	2	2	2	3	3	4	-1
K9a	1	1	2	2	2	2	-1
K9b	1	-1	-1	-1	-1	-1	-1
K10	2	2	1	1	1	1	16
K11b	1	1	1	1	1	1	-1
K12	84	81	81	81	81	81	81
K15	3	3	3	3	3	3	-1

Table 5. Set 1 limiting term frequencies (lower limit, keywords)

	F	K
Lovins	4	5
Porter	6	9
Snowball	6	9

Table 6. Different stem removal techniques (top 10)

5. Discussion

There were a number of queries for which queries on the indexes created from the identifier data produced no result at all, so we can hardly claim a great success. Nevertheless for a number of queries we did get a result, and it was often quite a good result. Furthermore, while basing our system on identifiers in source code did not obviously perform better than our previous system based on documentation, it did not perform obviously worse either. We believe this is a significant result — this is a strong indication that significant semantic information can be gleaned from the identifiers used in source code. We believe this is the first such indication.

There are several aspects of our study that means our results must be treated with some caution. The first is our choice of queries. It came from our previous study [11, 12], and that was based on a set by Ye to evaluate his system [29, 28], so there is some justification for our choice. Nevertheless it is easy to believe a different set of queries would produce different results. What we would really like to do is to gather standard set of "typical" queries made be develop-

	F	K
Term Frequency	6	10
tf-idf	6	9
Term occurrence	6	9
Binary occurrence	6	9

Table 7. Different term weighting technique (top 10)

ers. Not only would we get a wider set of components being asked for, but we would also get variations of wordings for the same queries. For example, instead of *judge if a file is a directory* we might get *is the file object a directory*.

While the corpus on which we have based our study is significantly bigger than those used in most other studies, in terms of the population of Java applications it is but a small fraction. Despite this, we were as successful (by some measure) as approaches using non-code artifacts. It would be interesting to see what effect a significantly larger corpus would have. It is somewhat surprising how insensitive our results were in terms of choice of data source so it is possible a larger corpus would not make much difference.

The number of non-results is something that we need to investigate further. One possibility is the lack of use of the relevant components, meaning we would not have any data on which to base the index on. We know from our usage study that much of the Standard API is not used in our corpus, so this is certainly a possibility. It does not explain some results, such as that for F4 however. The expected component here was `java.lang.Character`, which we would expected to be used. Furthermore, it seems unlikely that the methods needed to do capitalisation would be unused. One possibility is that the kinds of identifiers used in associate with `Character` used words that are not related to words like "capitalized", at least as far as WordNet is concerned. This means our results may be affected by the quality of WordNet's data.

A potential future direction is to combine the two approaches, as they do seem to complement each other. In particular, this would allow the issues associated with limited usage data that this approach presented here requires. This would at least involve using both the JavaDoc pages and the identifier data, but could also extend to either using the WordNet approach to index the JavaDoc information, or using LSI on the identifier data.

Finally we should mention our choice of performance measure. The standard performance measures for information retrieval systems are precision and recall, however they are not very useful for our case. Because we expect only a very small number of components to be relevant (we don't usually expect a library to provide the same functionality in many different ways) the precision is always going to be very small. Furthermore, as there is the possibility that there are no relevant components in the library, recall will not be a useful measure either. We consider evaluation based on the rank of the correct component to be most useful. Others have suggested rank-based measures may be more useful in other cases [20].

ID	Free Text			Keyword		
	Set 1	Lin	G	Set 1	Lin	G
1	**1**	33	-1	**1**	33	12
2	17	7	**1**	17	80	**1**
2b				**1**	9	17
3	-1	**2**	-1	-1	**2**	1
4	-1	**14**	-1	-1	**8**	-1
5	**8**	251	**7**	**8**	251	**7**
6a	**14**	1066	51	14	1600	**11**
6b	19	**2**	16	19	**2**	16
7	**5**	**5**	-1	**5**	**8**	66
8a	-1	324	-1	-1	130	**4**
8b	**2**	**3**	-1	**2**	**2**	1
9	**1**	69	-1	**1**	60	1
9b				**1**	26	7
10	**2**	**2**	-1	**2**	**2**	**2**
11a	-1	**24**	-1	-1	**21**	1
11b	**2**	**2**	-1	**1**	66	0
12	52	64	-1	52	62	**5**
13	-1	21	**5**	-1	**21**	0
14	-1	59	-1	-1	**20**	0
15	**6**	**3**	**1**	**3**	**3**	**2**

Table 8. Comparison with Lin and Google (G)

6. Conclusions

We have presented a study investigating the indexing of the Java Standard API using information gained from the identifiers used in the source code of applications that are used in conjunction with Java Standard API components. Extracting information from source code requires a reasonable amount of relevant source code — we use what we hope will be the beginnings of a standard Java corpus for this kind of study. Identifiers are often not English words themselves, and so we have had to decompose the identifiers. We use a heuristic based on a standard convention for Java identifiers and also identifying words that make up an identifier using WordNet [27]. To deal with the problems caused by synonymy and polysemy queries are processed using WordNet to create new queries that include words related to the original query words.

Our results, while not good enough to declare our approach a complete success, are very encouraging. For many of the queries we used we got be the expected component in the top 25 return results. This is at least as good as previous work. One important consequence of this is that our results provide what we believe is the first evidence that useful information can be extracted from source code.

Our approach clearly has a limitation in that it can only be used on the part of the Standard API that we have usage information for. Nevertheless our results are very encouraging. We also note that indexing based on documentation is also encouraging but not a complete answer. We suggest that the two approaches compliment each other, and believe that finding an effective means to combine them is the next challenge.

References

[1] M. W. Berry and M. Browne. *Understanding Search Enginers: Mathematical Modelling and Text Retrieval*. SIAM, 1999.

[2] J. M. Bradshaw. *An introduction to software agents*. MIT Press, Cambridge. MA, 1997.

[3] S. T. Dumais. Enhancing performance in latent semantic indexing (lsi) retrieval. *Technical Report 21236, Bellcore http://citeseer.ist.psu.deu/dumais92enhancing.html*, 1992.

[4] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1989.

[6] D. Harman. How effective is suffixing. *Journal of the American Society for Information Science*, 42:7–15, 1991.

[7] D. Hiemstra. A probabilistic justification for using *tf.idf* term weighitng in information retrieval. *Internation Journal on Digital Libraries*, 3(2):131–139, August, 2000.

[8] D. A. Hull. Stemming algorithms: a case study for detailed evaluation. *J. Am. Soc. Inf. Sci.*, 47(1):70–84, 1996.

[9] L. R. Jensen. A reuse repository with automated synonym support and cluster generation. Master's thesis, Department of Computer Science at the Faculty of Science, University of Aarhus, Denmark, 2004.

[10] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. In P. Willett, editor, *Document retrieval systems*, pages 132–142. Taylor Graham Publishing, London, UK, UK, 1988.

[11] M. Y. Lin. Supporting software component reuse with active-retrieving component repository system. Master's thesis, University of Auckland, 2005.

[12] M.-Y. Lin, R. Amor, and E. Tempero. A Java reuse repository for Eclipse using LSI. In *The Australian Software Engineering Conference*, Apr. 2006.

[13] H. Ma, R. Amor, and E. Tempero. Usage patterns of the Java Standard API. In P. Jalote, editor, *Thirteenth Asia Pacific Software Engineering Conference (APSEC06)*, pages 342–349, Bangalore, India, Dec. 2006. IEEE Computer Society.

[14] H. K. C. Ma. Using variable identifiers to index the Java 1.4.2 API. Master's thesis, University of Auckland, 2007.

[15] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991.

[16] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Massachusetts, 1999.

[17] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, volume 1, pages 138–150. NATO Science Committee, Jan. 1969. Presented at the NATO conference on software engineering, Garmisch, Germany, 7-11 October, 1968.

[18] A. Michail and D. Notkin. Assessing software libraries by browsing similar classes, functions and relationships. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 463–472. IEEE Computer Society Press, 1999.

[19] G. A. Miller. WordNet: An on-line lexical database. *International Jornal of Lexicography*, 1990.

[20] A. Moffat and J. Zobel. Rank-biased precision for measurement of retrieval effectiveness. Unpublished manuscript in submission, 2007.

[21] M. F. Porter. An algorithm for suffix stripping. In K. S. Jones and P. Willett, editors, *Readings in information retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[22] Qualitas Research Group. Qualitas corpus. `http://www.cs.auckland.ac.nz/~ewan/corpus/`, June 2007.

[23] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *Computer Graphics and Applications IEEE*, 10(4):76–82, Jul 1990.

[24] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[25] O. Steele. JWordNet. http://kwn.sourceforge.net, 1998.

[26] Sun Microsystems. Java 2 platform, standard edition, v 1.4.2, API specification. `http://java.sun.com/j2se/1.4.2/docs/api`, 2003.

[27] WordNet: a lexical database for the english language. `wordnet.princeton.edu`, 2007.

[28] Y. Ye. *Supporting component-based software development with active component repository systems*. PhD thesis, University of Colorado, 2001.

[29] Y. Ye and G. Fischer. Promoting reuse with active reuse repository systems. In *6th International Conerence on Software Reuse*, pages 302–317, London, UK, 2000. Springer-Verlag.

[30] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146–170, 1995.