**Automatically Invoking API Functionality Using Natural Language**

1. Abstract

To develop a new software, a programmer usually reuse existing functionalities provided by programming libraries. However, these libraries are normally large and complex. It’s difficult to the programmer find and use exactly the desired functionality. In this paper, to address these problems, we propose a novel approach to invoke functionality by using natural language. The approach’s objective is to map function call in natural language to programming language. There are two main tasks in our approach: Finding functionalities in the programming libraries related to programmer’s request, and generating function call statement in programming language for the found functionalities.

We have implemented our approach in an Eclipse plug-in and applied it on the Java standard library. Besides, we have also performed some evaluations to demonstrate that our approach is better than the other approaches in precision and recall.

1. Introduce

The programming libraries have grown more numerous and more diverse while the applications have grown more dependent on them. Thus, it’s difficult for programmers to find what functions they want and know how to call those functions. For instance, there are thousands of classes and more than 20,000 functions in the Java Standard Library (J2SE) [2], and more than 140,000 classes, functions in Microsoft .NET framework [3]. Several techniques [Koder, Krugle, Assimie,…] have been previously proposed to assist programmers in searching “appropriate functions for their tasks”, seeking “more guidance about these functions”, or identifying “function calls in sample code”. Some other techniques have tried to automatically generate function calls based on input-output type, such as Prospector [2] and PARSEWeb [5].

Unfortunately, a programmer normally has a tendency to describe their requests in natural language. While most functions in software libraries are represented in a structured form of a programming language which is hard to understand. To close the gap between programming language and natural language, API specification (API spec) is applied in many nowaday researches, such as Mica [3] and Exemplar [6]. Nevertheless, Mica uses API documentation to refine the results of the search, while Exemplar uses API documentation as an integral instrument to expand the range of the query.

In our earlier work [] (previous work), we developed a function search engine, call FSE, to help programmer find function. However, one of limitations of FSE is result returned very large. Therefore, in this work, we apply approaches of keyword weighting to improve the result. The most common weighting method is Term Frequency - Inverse Document Frequency [42] (TF-IDF) which find the most important keywords for the document within a corpus by assessing how often the keyword occurs within the document (TF) and how often in other documents (IDF). With categorized documentation like API spec, some approaches such as Odds Ratio [30], Information Gain [52], and Chi-squared [52] which calculate the score based on certain features belongs to given categories are applied. A novel approach, call Bi-Normal Separation (BNS), is applied in [14]. The idea of BNS is to compare the two distributions; the larger the difference between them, more important the feature.

In this paper, we propose a novel approach to automatically call functions in natural language by compounding the above approaches. Our main idea is allow programmers describe their needed functionalities in short natural language queries while programming their code. After that, a list of function suggestions retrieved from API spec is returned. Then, the programmers might choose suggestions related to their tasks. With each suggestion, a function call is generated with some empty positions. Finally, we also define some Heuristic to assign variables in the declaration of the souce code to the empty positions in this invocation.

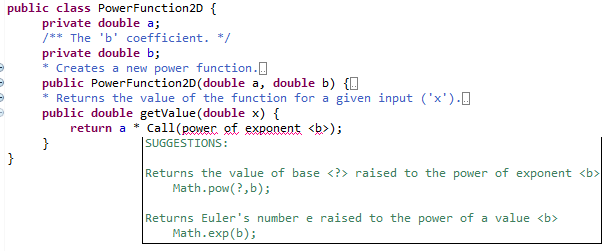


Figure 1.

Figure 1 is a example for our idea. In this figure, a programmer needs to raise a certain value by an exponent <b>. In editor, the programmer might enter a following simple description into source code: “power of exponent <b>”. After that, he received two suggestions related to his request. With each suggestion, the variable <b> is inferred to fill into the natural language guidance and invocation. Based on the guidance, the programmer might choose the suitable functionality for their task.

This paper has 4 contributions:

* Proposing the novel keyword weight method to match natural language queries to the function descriptions based on the structure of API spec. This method is efficient in reducing redundant results.
* Proposing the novel ranking method with three score levels: Class definition score, function definition score, parameter score. This method is faster and more efficient than our previous ranking method [].
* Proposing the novel method to generating automatically function call statement by inserting variables into correct positions of the function call. Note that, the variables are retrieved from declaration in programmer’s source code which contain these function calls.
* Proposing the novel method to deal with problem of type compatibility by mining content *inheritance hierarchy* in API spec.

The paper has five main contents. The first content presents the method to preprocess API spec and user’s query. The second content introduces our function call generation approach in detail. In the third part, we show our plug-in …

1. Preprocessor

In previous work [], we applied a simple approach to search function definitions from API spec based on some natural language processing technologies. However, the limitation of this method is the large result returned. According to our statistics, in over 3500 nouns and 500 verbs retrieved from API spec, there are about 2700 nouns and 400 verbs appeared less than 10 times. Only 6 nouns and 3 verbs appear more than 900 times, there are *string*-1111 times, *set*-1203, *method*-1344, *component*-1594, *value*-2579, *object*-3410, *get*-901, *set*-1559, *return*-5898. Thus, if the special keywords (such as *get*, *set*, *object*, *value*,…) are used to search, the return result will be really large. For example, if we query “*returns a char from a string*”, about 7000 functions containing the *return* or *string* keywords are returned. Therefore, to reduce size of the keyword set, we applied a method of multi-level assessment based on the work by Wan and Xiao [48]. The main idea of the approach is to limit the number of retrieved functions in irrelevant classes.

This section is organized as follows. Subsection A presents structure of API specification. After that, based on this structure, a method to mine API spec is applied in subsection B. Finally, to improve performance in information retrieval, we even use some natural language processing (NLP) technologies to expanse user’s query in subsection C.

* 1. API specification structure

API specification is a categorized documentation with multi-level. Figure 2 shows the structure of API specification with three levels: Class level, function level, and parameter level. The top level related to classes (and interface) definition, such as: Class description, specification, inheritance hierarchy, and a set of function definitions. In the lower level, there are contents of function (or method) definitions in a class: Function description, specification, return value, exception, and a set of parameters. The lowest level is some optional contents related to parameters in each function: Parameter name, type, description.

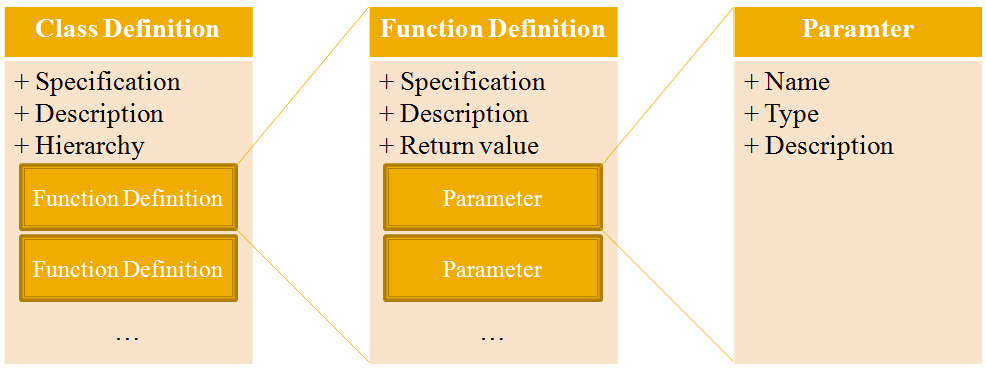


Figure 2

It’s easy to identify that every level contains a description in natural language (class description, function description, parameter description). Thus, some NLP technologies are applied to retrieve keywords from these descriptions. The first is the **stopword filtering** technology used to remove stopwords (such as *the*, *is*, *at*, *which*, *on*,…) from API spec’s discriptions. **POS tagging** (part-of-speech tagging) is the technology to mark up a word in a natural language sentence (NL Sentence) as corresponding to a particular POS based on both its definition and its context. **Stemming** is the technology to reduce inflected (or sometimes derived) words to their root form, for instance, the verb *return* is the root form of words “*returns, returning, returned*” after stemming to remove suffixes such as “*-s, -ing, -ed*” respectively.

Example: For the function description “Gets an element in the collection”. The followings are results of the above stages. **Stopword Filtering**: “*Gets element collection*”. **POS Tagging**: “*Gets/VB element/NN collection/NN*”. **Stemming**: “*Get element collection*”.

Besides, we also retrieve keywords from some other non-natural language contents, such as name of class or function. Then, in the next stage, the retrieved keywords are weighted. After weighting, some keywords with very small weight are removed.

Moreover, in this paper, we even mine the content *inheritance hierarchy* in class definition to address issue of type compatibility when assigning variables (in query) to parameter (in function definition).

* 1. Keyword weighting

**Term Frequency - Inverse Document Frequency** is the most traditional keyword weighting method used in information retrieval. In API spec, each class definition has a description in natural language, so we can use the TFIDF method to weighting keywords in this content. The idea is to find the most important keywords in every class definition by assessing how often the keyword occurs within the class description (TF) and how often in other class descriptions (IDF):

Where *tf*(*k,c*) is the term frequency of word k within the class description *C*, *|C|* is the length of the class description, *df*(*k*) is the number of class descriptions which the word *k* appears in, and *N* is the number of classes in the API spec.

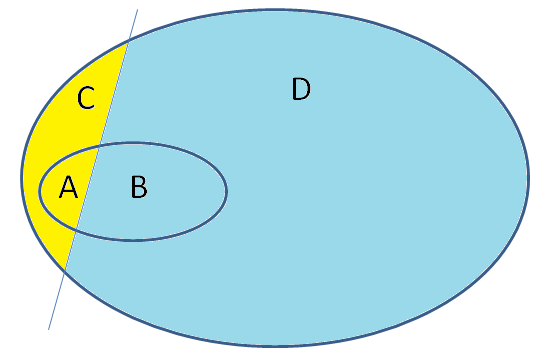


Figure 3

However, API spec is a structured documentation. A class definition contains not only class description content but also a set of function definitions. Thus, this structure is mined to make more effective for our keyword weight method. In our observation, in a class definition, the more function definitions which a keyword appears in, the more important the keyword should be. Suppose that, c is a class definition in API spec. Based on class definition c, API spec is divided into four part as shown in figure 3: Part A is the number of function definitions in c containing k, part B is the number of function definitions out of c containing k, part C is the number of function definitions in c without k, part D is the number of function definitions with neither c or k. In table 1, we propose some primary feature-based scoring methods, note that N is the total of function definitions and F-1 is the inverse Normal cumulative distribution function.

Table 1

|  |  |
| --- | --- |
| **Keyword Weighting Methods** | **Mathematic Formations** |
| Odds Ratio (OR) |  |
| Mutual Information (MI) |  |
| Information Gain (IG ) |  |
| Chi-squared (­X2) |  |
| Bi-Normal Separation (BNS) |  |

**Bi-Normal Separation (BNS)** is the effective approach proposed by Forman. This approach uses the standard normal distribution’s inverse cumulative probability functions of positive samples and negative samples. To avoid the undefined value such as F-1(0), Forman limited both distributions to the range [0.0005, 0.9995]. In other work [], Forman compared the performance of BNS against several other feature weighting approaches including Odds Ratio, Information Gain, and ­ Chi-squared [15]. In his experiments, BNS produced the best results with Information Gain performing the second best. In our approach, the BNS method is chosen to improve the quality of keywords.

Finally, we propose a keyword weighting method to score all words in a class C by combining TFIDF and BNS:

Where *p* (0 ≤ *p* ≤ 1) indicates the weight that is used for giving more emphasis to either TFIDF or BNS score. However, to remove some keywords with very small score, we use a threshold t to select a list of top keywords that have a score above t. Threshold t is decided by user.

* 1. Query expansion

Query expansion is the process of reformulating a query to improve performance in information retrieval. User’s query is in natural language, so we also use the NLP technologies presented in the previous subsection to retrieve keywords. Especially, in this subject, we even build a set of user-defined synonyms for the retrieved keywords based on our knowledge. For instance, *append*-*add*-*insert* are synonymous in our scope.

Especially, in this subject, programmer can fill variables into query. After that, based on parameter contents in function definition, these variables are located to suitable position in function call statement. Our observation detect that programmer has a tendency to set the variable into some fixed locations in the query. To detect the fixed locations, we performed a study on programmer’s requests. In this study, we provided a list of queries without variables. With each query, we gave some variables with their functionality descriptions, and requested programmer to fill these variables into the suitable positions in the query. Based on the result of our study, we see that most variables were located at the end of the noun phrases.

For example: Suppose that we have two variables declared as follows “*ArrayList l; Object o;*”. Programmer is requested to fill these variables into the query “*Inserts a specified element into a sorted collection*”. The result we collected is: “*Inserts a specified element* ***<o>*** *into a sorted collection* ***<l>***”.

Thus, in this subsection, we propose a method to retrieve variable’s noun phrase based on query’s context. Then, the noun phrase are compared to parameters (in function definition) to identify the proper position of this variable in function call statement. In the first stage in the method, we use a natural language parser to change the query into a parsing tree. After that, it base on noun phrases in this tree to know exactly whether a keyword relates to a variable or not.

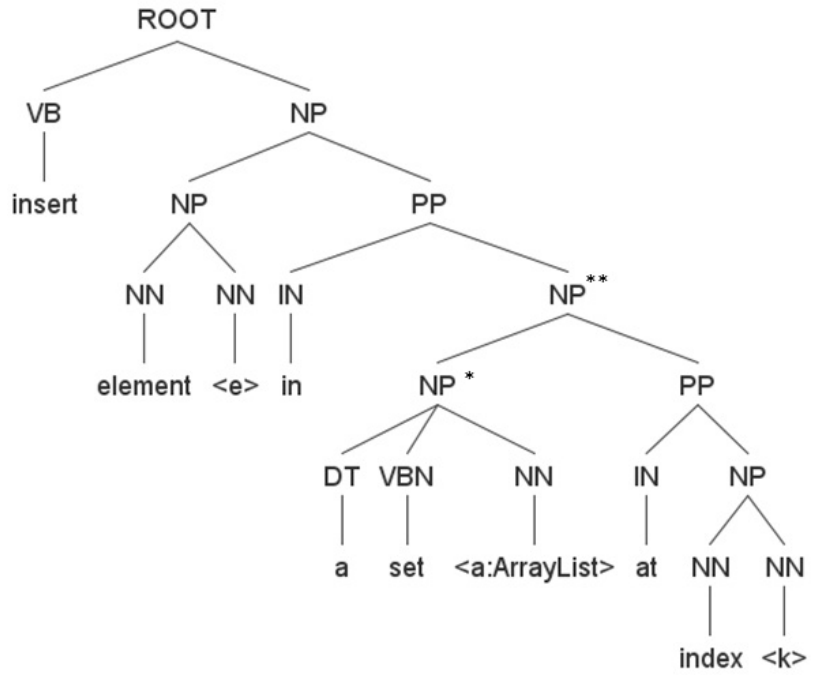


Figure 4

For example: In the figure 4, a query in natural language is parsed in a tree structure by using Stanford-Parser tool [16]. In this tree, the keyword *element* and variable *e* are same noun phrase (NP), so they relate to each other.

1. Our approach

Figure 5 shows an overview of our approach with three stages. The first stage is to look up class definitions, which relate to user’s query, from API spec. Secondly, finding the suitable function definitions from these classes. Finally, generating function calls based on contents of the found function definitions.



Figure 5

This section is organized as follows. In subsection A, classes which are relevant to user’s query are retrieved from API spec. Subsection B proposes the novel approach to calculate degree of fitness between function definitions and user’s query by using WordNet. Subsection C shows the heuristic to infer position of variable in function call.

* 1. Class retrieval

In the first stage, the vector space model is applied to calculate the similarity between class definition and user’s query. In this model, every class definition C in API spec and the query Q are represented as vectors of index keywords. After that, the cosine of the angle between vector Q and C is calculated:

Where (*C.Q*) is the intersection of the class definition and the query vectors, ||*C*|| is the norm of vector *C*, and ||*Q*|| is the norm of vector *Q*. The weight vector for class definition *C* is {*Sc*(*k1,C*), *Sc*(*k2,C*), …, *Sc*(*kn,C*)}, with *Sc*(*k,C*) is calculated as follows:

Where *W*(*k,C*) is keyword weight presented in previous section. *WPOS*(*k*) is the word evaluation based on part-of-speech tags (POS-tags). As we know, in natural language, if words belong to different POS types, they have different degrees of importance. In natural language, verbs are often defined as words which show actions or states of being. The verb is the heart of a sentence - every sentence must have a verb. Recognizing the verb is often the most important step in understanding the meaning of a sentence. In the rest of POS types, nouns are ranked higher than others. On the contrary, prepositions, pronouns, conjunctions, and interjections are often defined as stopword which need to be removed.

* 1. Function retrieval

After the first stage, we collect a list of class definitions. In this stage, the query Q is compared to every function definition in the class definitions. Suppose that F is a function definition in C. Some NLP technologies such as stopword removing, POS tagging, stemming are applied to retrieve keywords from F and Q. Assume q is a keyword retrieved from Q and f is a keyword retrieved from F (extracting from function name and description), the similarity *Rf*(*F*,*Q*) is calculated as follows:

Where L(F) is the length of F, L(Q) is the length of Q. *Sf*(*f,q*) is value of similarity between f and q:

We use the dictionary WordNet to calculate this value *Sf*(*f,q*). In WordNet, words are grouped into sets of synonyms called synsets. It mean that the words belonging to the same synset have the similar meaning. Besides, if a synset indicate a more general meaning than other synset, this synset is a hypernym of the other. For instance, synset {parent} is a hypernym of synset {mother, female parent}. Moreover, if *A* is a hypernym of *B* and *B* is a hypernym of *C*, *A* is a hypernym of *C*. The distance *D*(*A,C*) = 2.

* 1. Code generation

After previous subsection, we collect a list of suitable function definitions. In this subsection, the content *parameter* of function definition is applied to map user’s query (which is function call in natural language) to function call statement in programming language. As we know, the standard syntax of a function call statement is defined as follows:

* *o*.*funName*(*arg1, arg2, …, argk*) for non-static function, or
* *className.funName(arg1, arg2, …, argk)* for static function.

Where *funName* is name of function, *className* is name of class, *o* is an object which invokes function *funName*, and *argk* is an argument. Note that, every argument is linked to a corresponding parameter in function definition *funName*. Thus, basing on the parameter information (such as parameter name, type, and description), we can infer variable positions in the call statement. However, we have some following problem.

In the first problem, there are two kinds of function: Static function and non-static function. With the static function, we assign directly the variables (in query) to parameters (in function definition), for instance, the result of the query “to the power of an exponent *<e:double>*” is a static function call “*Math.exp*(*e*)”. With the non-static function, before filling variables, we need to identify the variable of object *o* which calls this function, for instance, the result of the query “inserts an element *<e:Object>* in a list *<l:ArrayList>*” is “*l.add*(*e*)”, with *l* is object *o*. Thus, to address this problem, the object *o* is matched to a special parameter in function definition, where the parameter type is the *return type*, and the parameter description is the *function* *description*.

Another problem come from type compatibility. Normally, if a variable and a parameter have different type, they are not relevant to each other. However, in some special situation, if a variable type is inherited from a parameter type, the variable is relevant to the parameter. Therefore, to address the problem, the content *inheritance hierarchy* in class definition is applied to check compatibility between variable and parameter.

With a function definition, there are many options to fill the variables into the parameters. For a function definition with n parameters and a query with k variables. Suppose that O = {*v1/p1,v2/p2,…vk/pk*} is a option in mapping variables to parameters, where {*v1,v2,…,vk*} is a permutation of k variables, and {*p1,p2,…,pk*} is a combination of n parameters. Note that, the number of parameters in a function is often less than 4, so the number of combinations of parameters is small enough not to affect performance. However, to limit the number of options, we remove all the options containing any type incompatibility pair vi/pi. For each option, degrees of fitness are given by:

Where *Sp*(*vj/pj*) is the number of keywords which are similar between variable *vj* and parameter *pj*. Note that, the keywords of *pj* are retrieved from its name or its description, the keywords of *vj* are presented in the previous subsection “*query expansion*”.

The declaration in source code:

Integer v = 255;

Integer hex = 16;

Function call in natural language:

Convert an integer <v> to a string representation in a radix of <hex>

Here is a suitable function definition found:

java.lang.Integer: static String toString(int i, int radix)

i - an integer to be converted to a string.

radix - the radix to use in the string representation.

Some function call options are suggested:

Integer. toString (v, hex);

Integer. toString (hex, v);

Figure 6

After calculating the degrees of all options for the function definition, the option with highest degree is chosen to make the desired function call. For instance, in the figure 6, a programmer need to convert the decimal value of 255 to hexadecimal. Suppose that, this programmer declared two variables *<v>* and *<hex>* in his editor. In our approach, the programmer can enter a function call in natural language in his source code. After that, a list of suitable function definitions returned. Assume that the function *toString* is one of results of this list. There are two options to assign variables *<v>* and *<hex>* to the parameters *i* and *radix* in this function. The results of these options are: R({*v/i, hex/radix*}) = {integer, string, representation, radix}, R({*hex/i, v/radix*}) = {string}. Hence, the code of this function call statement is: “*Integer*.*toString(v,hex);*”.

Similar to other function definitions, each of them make a corresponding function call. The final result is a list of function calls. Note that, the number of the generated function calls can be smaller than the number of the found function definitions because some function definition don’t have any relevant options.

In the next stage, the list of the function calls is ranked in descending order of appropriate degree of query. The following is the compound formula of ranking the degree between the function call statements and the query.

or

1. IMPLEMENT

In this paper, we introduce our framework to support programmer to call function in natural language. The framework is a plug-in of Editor. The followings are the main stages in the framework’s infrastructure:

* Extracting data from Editor, consist of query and list of variables.
* Using some presented NLP technologies to preprocess API spec and user’s query.
* Retrieving a list of suitable class definitions from API spec.
* Finding suitable function definitions in the list of the retrieved class definitions.
* Generating function call for every found function definition

Chưa xong

[1.] Recovering Traceability Links between an API and Its Learning Resources

[2.] Inferring Resource Specifications from Natural Language API Documentation

[48] X. Wan and J. Xiao. CollabRank: Towards a collaborative approach to

single-document keyphrase extraction. In Proceedings of 22nd International

Conference on Computational Linguistics (COLING’08), United Kingdom,

pages 969–976, August 2008.

[14] G. Forman. An extensive empirical study of feature selection metrics for text

classification. Journal of Machine Learning Research, 3:1289–1305, 2003.

[42] G. Salton and C. Buckley. Term-weighting approaches in automatic text

retrieval. Information Processing and Management, 24(5):513–523, 1988.

[15] G. Forman. BNS feature scaling: an improved representation over TF-IDF

for SVM text classification. In Proceedings of the 17th ACM Conference on

Information and Knowledge Management (CIKM’08), USA, pages 263–270,

October 2008.

[52] Y. Yang and J.P. Pedersen. Feature selection in statistical learning of text

categorization. In Proceedings of the Fourteenth International Conference on

Machine Learning (ICML’97), USA, pages 412–420, July 1997.