Querying Source Code with Natural Language

Markus Kimmig, Martin Monperrus, Mira Mezini

Querying Source Code with Natural Language

Markus Kimmig
Technische Universität Darmstadt

Martin Monperrus
University of Lille

Mira Mezini
Technische Universität Darmstadt

***Abstract*—One common task of developing or maintaining software is searching the source code for information like specific method calls or write accesses to certain fields. This kind of information is required to correctly implement new features and to solve bugs. This paper presents an approach for querying source code with natural language.**

I. Introduction

The ability to correctly find source code elements is crucial for many software engineering tasks [1]. For instance, a programmer may ask "Where is the field balance read?" before changing the way it is set, in order to avoid undesired side-effects and regression. Providing tool-support for searching source code is especially important when the code base is huge or when developers are unfamiliar with it.

In this paper, we present an approach that enables the developer to query source code with natural language. Contrary to [2], our approach does not impose rules on how the queries have to be stated. It only assume that the query is grammatically correct English. It combines Part-of-Speech Tagging, statistical analysis and a touch of embedded knowledge to identify which words of the query describe what the developer is looking for. Eventually, the natural language query is translated into a call to a code query engine. Consequently, the scope of supported query kinds is given by the underlying code query engine, and our approach handles the multitude of variations and modulations in ways of expressing queries. Our prototype implementation is built on top of the Eclipse JDT code query engine [3] and the user-interface is tightly integrated with the Eclipse development environment.

The approach is evaluated with a user study of 14 subjects. The experiment consisted of a number of common programming and refactoring tasks which require to find an appropriate piece of code in an unknown project. Subjects were only allowed to use our prototype system to find the correct piece of code. A comprehensive log of all queries from every subject shows that our system is able to correctly understand 83% of all subject queries. We also interviewed every subject to get detailed feedback about the prototype as well as possible future improvements. The interviews show that the tool is mostly perceived as being a valuable addition to the toolbox of software developers.

II. From Natural Language to Formalized Queries

In our approach, when the developer has a development task that requires examining existing source code elements, (s)he expresses a query in natural language. A translator analyzes the query and translates it to a formalized query for a code query engine. The results given by the code query engine are displayed in the development environment, and the developer starts to examine the code elements found or refines/reformulates his query if (s)he has not found what (s)he is looking for. It is also important to give the developer a way to inspect the result of the translation process to help him/her assess that the translation was correct.

Our approach to understanding natural language queries over source code consists of 7 sequential steps that are presented below. The full process is summarized figure 1, along with a concrete example.

a) Cleaning and Tokenizing the Query: The first step is to tokenize the query. Tokenization means that the string representing the query is split into tokens which each represent a single word of the query. We use a tokenizing function based on a regular expression that splits the query string at one or more white spaces.

For example, the query "*Which methods return type integer*" yields the following tokens: *Which*, *methods*, *return*, *type* and *integer*. The order of the tokens is kept, because it is an important piece of information that we use in the next steps (see section II-0g).

b) Part-of-Speech Tagging of the Query: A Part-of-Speech (POS) Tagger is an algorithm that assigns a grammatical category (e.g. noun or verb) to every word of a sentence. For instance, the query "*Where are instances of type Integer created*" could be POS-tagged as follows: "*Where(question word), are(be), instances(noun), of(preposition), type(noun), Integer(noun), created(verb)*". POS-tagging the query enables us to enrich the query with grammatical information that we will use later for inferring the code query engine parameters.

In the following, the ordered list of important grammatical types (using only nouns and verbs) is called the grammatical form of the query. For instance, *noun -> noun -> noun -> verb* is the grammatical form of the previous query.

c) Selecting Code Search API Parameter Candidates: A code query engine can have different parameters. Some parameters must take a value in a finite range of possible values, some are free. Our prototype uses the Eclipse JDT code query engine. The core of the JDT query engine consists of an API call taking three arguments as parameter: the *element kind* is the kind of source code element being sought (e.g. type or method); the *code context* is the programming context in which we are interested in. For instance, if the kind is "method", the context may be "method call" or "method declaration"; the *identifier* is a arbitrary string expression describing the element that is sought (e.g. "Integer" or "toStr*").

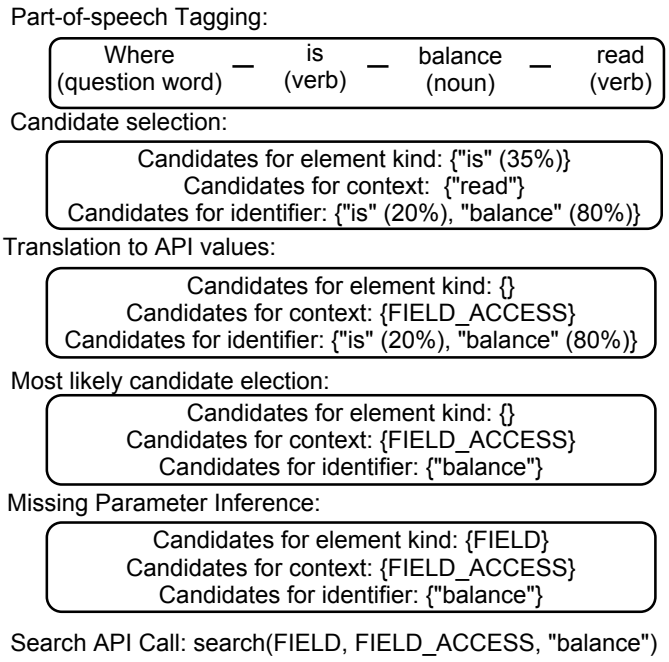

Fig. 1. Example Sequence of the Transformation of a Natural Language Query to an API Call with Valid Values.

Let us assume the user entered the query "*Which methods take a parameter of type Integer*". We can deduce that the user is searching either a method or a type, hence they are two candidate values for the code search parameter "element kind": METHOD or TYPE. For identifiers (which are free text), any word of the query would be a valid parameter value, hence the 8 words of the query are possible candidates. We describe in this section how we use the query words, the word order and the POS-tagging information to select those candidate values.

Our selection mechanism is inspired by naive bayes classification [4]. We first tag a set of training data to indicate which part of the query corresponds to which search API parameter, then we compute the probability of the relationship between each piece of query information and a search parameter value.

d) Training data: The training data consists of queries which were manually annotated with information about which tokens correspond to which search API parameter. For example, a single line from the training data is: "*What methods return:context type:kind_of_sourcecode_element Integer:expression*". It contains three annotations consisting of a pair "word:tag": *return:context* declares the first verb ("return") to be the context parameter value, *type:kind_of_sourcecode_element* declares the second noun ("type") to be the kind of source code element being sought and *Integer:expression* declares the third noun ("Integer") to be the expression parameter value for this query.

Annotated queries are defined based on a combination of the implementor's expertise and real-user data collection. For replication, ours are published on [5]. According to our experience, the number of required training data to obtain a good performance is around 250 queries. The reason for this relatively small number is that while the query space is infinite, the space of different grammatical query forms (as given by the POS tagger using only nouns and verbs) is finite, and most queries can be described by a few dozens of grammatical forms.

e) Candidate Selection: When a developer enters a code search query, we first compute its grammatical form. Then, we search the training data for annotated queries whose grammatical form (consisting only of nouns and verbs. see II-0b) corresponds to the query form. In other terms, we search the training data for queries that match the grammatical form of the POS-tagged query. If one is found, we add to the candidate list for the respective parameter the words from the query matching the POS-tags and positions of the accordingly annotated words from the training data. For instance, let us assume that a training query is "*Where is field::kind_of_sourcecode_element balance read?*". It contains one tuple of annotation saying that the first noun (field) corresponds to the element kind. Let's now consider the query "*Where is class Widget used?*". Since it has the same grammatical form, we add the noun "class" to the candidate list for the element kind.

To decide which candidate to use, we compute a probability value for each word of the query which describes how likely this word is a candidate for the search API parameter. Note that although we use only nouns and verbs to match the query against the training data, any word of the query can be a candidate for a parameter, regardless of its POS-tag, for example the second adjective.

For instance, in the query "*Which methods take a parameter of type Integer*", the grammatical form is *noun -> verb -> noun -> noun -> noun*. Let's assume that the training data contains 10 queries with the same grammatical form. In 8 of them, the first noun refers to the code element kind. Hence, the probability of "methods" to indicate the code element kind is 80% (8/10). However, in two queries, the third noun represents the code element kind. In this case, the resulting candidate list for the sought element kind is: { method (80%), type (20%)}. To sum up, for a given query we compute NxM probabilities where N is the number of words of the query and M is the number of parameters required by the underlying code search API (e.g. M=3 for the JDT code query engine). Probabilities are often equal to zero because many combinations have no corresponding annotations in the training queries with a matching grammatical form.

f) Translating to Concrete API Candidates: We now have a set of candidate words for each search API parameter. Some API parameters are free text, which means that we can pass the user query word as is. However, most search API parameters must fit in a fixed enumeration. For instance, the kind of sought source code elements could be either {METHOD, CLASS, PACKAGE, ...}. However, the user may use different syntactical forms (e.g. plural) and synonyms. (e.g. "function" for "method").

For each search API parameter which is not free text (e.g. the element kind that is sought), we define a mapping from a list of words in stemmed form to API parameter values. For example, if the user enters *methods* (stemmed into "method")

in the query and this word is a candidate for the source code element kind, the mapping translates it to the valid parameter value *METHOD*. Note that multiple words may be mapped to the same parameter value in order to allow the user to refer to the same programming concept with multiple words (for example *class* and *type*). The mapping data is defined once at implementation time, with the keys being stemmed words and the values being the corresponding search parameter values.

Some valid API values must sometimes be deduced by a combination of words. For example if the query contains the substring "*super reference*", this hints that we are looking for method calls to the parent classes (i.e. calls to super). We add a tuple of words to candidate lists when annotated queries contains several annotations for the same parameter. Those tuples of words may have a corresponding entry in the mapping data that we call a multi-key. As for simple keys, multi-keys are mapped to a single search API parameter value. In the previous example, there is a mapping from the tuple $< super, reference >$ to the search API parameter "SUPER_REFERENCE_METHOD_CALL".

g) Electing the Most Likely Search API Parameters: To elect the most likely value for each for each search API parameter, we define several rules.

Rule #1: If a candidate word has no translation to a valid parameter value in the mapping, it is removed. For example, if *Integer* is a candidate for the kind of source code element parameter, but the mapping does not contain a translation to any kind of source code element parameter value, the candidate word *Integer* is removed from the candidate list for sought element kinds.

Rule #2: Sometimes candidate words are part of single key and multi-key translations. Take for example the query "*Where are parameter bounds of type Integer*". There are two candidates for the context parameter value: *parameter* indicating the API parameter value METHOD_PARAMETER and the multi-key candidate *parameter bounds* indicating the API parameter value PARAMETER_BOUND (for example Array<I extends Integer>), which overlap in the query. In these cases, we always choose the multi-key candidate with the highest number of words, because it is the most specific.

Rule #3 If several candidates remain after rule #1 and #2, the chosen API parameter value is the one with the highest probability computed from the training queries (see II-0e). If no value could be found, the value is set to unknown.

Rule #4: We always start by electing the context parameter value, then we elect the code element kind, and then the identifier. Starting with the context parameter has two rationales. First, according to our experience, this parameter is correctly chosen with the highest reliability. Second, it enables us to infer the correct element kind is many cases, as shown below in II-0h.

Rule #5: If a candidate has been chosen as the final parameter value, all other candidates with the same token are removed from the other candidate lists. For example, let us assume that the element kind candidate list consists of the first noun, and that the context candidate list contains the first noun and the second verb. If the first noun is elected as the value for the context parameter, the first noun is removed from all other candidate lists.

Rule #6: The last parameter value to choose is the identifier (a free string). We first check whether a word is indicated as such with double quotes in the input query (e.g. *Where is declared metho "printToConsole"?*). If this is the case, the identifier parameter is set to this quoted word. If this is not the case but there is exactly one word in the query which contains wildcard characters (* or ?), this word is chosen. Otherwise the word with the highest probability to be correct is chosen (see rule #3).

h) Inferring Missing Values: At this point, for each search API parameter, we have either a unique valid value or nothing. To be able to process queries that provide incomplete information, we also store the information of the dependency between search API parameter values. For instance, the context parameter often implies one specific kind of source code element to be sought. Take for example the following query: "*Where is number read*". This query does not contain any explicit information on the kind of source code element that is sought (e.g. type, method, etc.). However, the word "read" indicates the context parameter value *READ_ACCESS*. With this information, we are able draw the conclusion that the only possible kind of source code element parameter value is *FIELD*, because all other code elements, like for example *package* or *method*, can not be read.

i) Performing the Code Search: Figure 1 sums up our algorithm to translate a natural language query into a set of valid code query engine parameter values. The final step eventually consists of using the target search API to carry out the search request with the parameter values determined by the presented strategy. The search results are then displayed to the developer. Our prototype is integrated into the Eclipse IDE as a plug-in. A screenshot is shown in figure 2.

III. Evaluation

To evaluate our system, we conducted a user experiment. We recruited 14 subjects to perform 13 code maintenance tasks which require navigating over source code (e.g. *"Method init() is called in a method where it doesn't make any sense. Remove the method call.*)". The experiment takes about 30 minutes per subject, and consists of three phases. First, the experimenter presents the experiment and the prototype to the subject. Then, the subject works on the given tasks for about 15 to 20 minutes. The experiment concludes with an oral and recorded interview. Subjects are students in computer science at our university

The subjects can only use the Eclipse IDE to complete the tasks. The only IDE feature they are allowed to use is the natural language search tool. In particular, to browse and find code, they are not allowed to use the "package explorer view" and the "outline view", the "open type widget", the text and java search widgets. Indeed, the goal of the experiment is to find out how well our approach is sufficient for finding code elements.

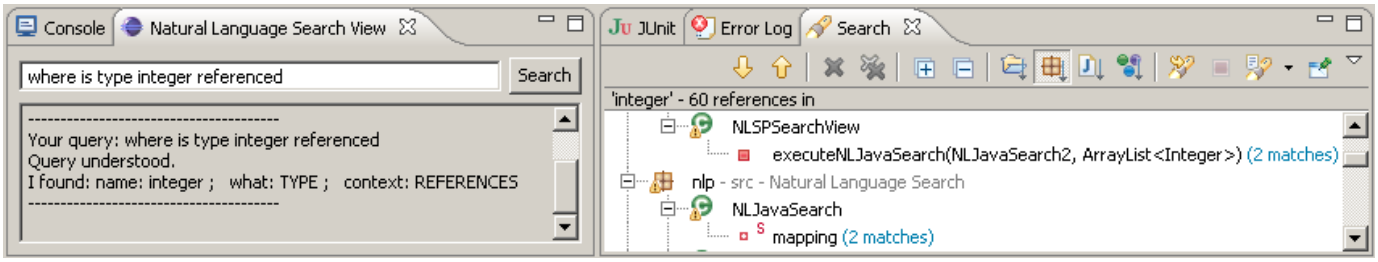

Fig. 2. Screenshot of our Prototype Implementation in the Eclipse IDE. The graphical interface contains a query field, a feedback area, and a search result area.

From all 14 sessions, we computed the number of understood queries and the number of correctly understood queries. These have to be separated, because the fact that the tool understood a query only means it found a value for all three parameters, but not necessarily the right ones. For example, if the user enters the query "*Where is method init() called*" the tool might choose *method* for the "search for" parameter, *call* for the context parameter and *where* for the expression parameter, which is wrong. The correct values would be be METHOD, CALL, "init()". The average percentage of understood queries was 91% (251/276). The average percentage of correctly understood queries was 83% (228/276). However, about 91% (229/251) of all understood queries were understood correctly. We had 5 subjects had previous experience with the Eclipse code search widget, 4 of them stated they would prefer working with our code search system rather than the Eclipse one. As future work, we plan a controlled experiment to thoroughly compare both systems.

IV. Related Work

A large body of work has been done on developing systems and frameworks to query source code. There are several approaches that use specialized query languages to retrieve information about source code (e.g., [6], [7]. These approaches require learning the syntax of the query language before being able to work with the tools, while with a natural language interface there is almost no learning phase.

Queries are not only over source code. For instance, de Alwis and Murphy [8] described different software maintenance queries. Hill et al. [9] uses NLP based on program identifiers to improve contextual code search (which pieces of code are about this topic?). Ko et al. [10] presents a system for querying program output, and not source code itself. Wang et al.'s approach [11] detects duplicate bug reports using NLP.

Würsch et al. [2] described a powerful system that is similar to ours. Our technical contribution describes a completely different algorithm using incomparable techniques. While they use tools from ontology engineering, we use natural-language processing techniques. Our user study also contributes with first insights on how developers react on using such systems. But apart from these technical differences, our approach is novel in the sense that it supports completely free queries, while theirs is based on guided input, i.e. the developers select a query into an adaptive list of possible queries. This new degree of freedom creates a whole new challenge, and our paper aims at contributing to this new research direction.

V. Conclusion

We presented our approach for querying source code with natural language queries. The approach translates natural language queries to concrete parameters of a third party code query engine. Our approach uses a combination of natural language processing techniques (Part-Of-Speech tagging, stemming), as well as a custom algorithm that extracts statistical data from manually annotated training queries. Our prototype implementation uses as underlying code query engine an unmodified off-the-shelf version of the Eclipse JDT code query engine. We evaluated our approach using a user-study with a total of 14 subjects. Our prototype was able to correctly understand 83% of queries: 91% of 276 queries which have been entered by subjects were recognized and 91% of them correctly. Future work consists of conducting a controlled experiment to compare our approach against related systems on the same set of tasks.

References

[1] G. C. M. J.Silito and K. D. Volder, "Questions programmers ask during software evolution tasks," in *Proceedings ACM SIGSOFT Symposium Foundations of Software Engineering*, pp. 23–34, 2006.

[2] G. R. M. Würsch, G. Ghezzi and H. C. Gall, "Supporting developers with natural language queries," in *Proceedings of the International Conference on Software Engineering*, pp. 165–174, 2010.

[3] Eclipse Foundation, "Java development tools (JDT)." http://www.eclipse.org/jdt/, 2011. Accessed March 1, 2011].

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] M. Kimmig, M. Monperrus, and M. Mezini, "Replication data." http://www.monperrus.net/martin/ase2011, 2011. Accessed May 9, 2011.

[6] M. V. E. Hajiyev and O. de Moor, "Codequest: Scalable source code queries with datalog," in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 2–27, 2006.

[7] D. Janzen and K. de Volder, "Navigating and querying code without getting lost," in *Proceedings of the International Conference on Aspect-oriented Software Development*, pp. 178–187, 2003.

[8] B. de Alwis and G. C. Murphy, "Answering conceptual queries with ferret," in *Proceedings of the International Conference on Software Engineering*, pp. 21–30, 2008.

[9] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the International Conference on Software Engineering*, 2009.

[10] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *Proceedings of the International Conference on Software Engineering*, 2008.

[11] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the International Conference on Software Engineering*, 2008.