HAL
open science

A Natural Language Interface for Code Search

Markus Kimmig, Martin Monperrus, Mira Mezini

To cite this version:

Markus Kimmig, Martin Monperrus, Mira Mezini. A Natural Language Interface for Code Search. [Technical Report] hal-01094267, TU Darmstadt. 2011. hal-01094267

HAL Id: hal-01094267

https://hal.science/hal-01094267v1

Submitted on 12 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Natural Language Interface for Code Search

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 a natural language interface. It enables the developer to execute a huge range of precise searches while being as easy and intuitive to use as writing natural language. The evaluation shows that the prototype implementation, integrated with the Eclipse development environment for Java, supports a wide range of queries and is able to correctly understand most real developer queries.**

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. Table I lists a sample of software development tasks and related queries. Providing tool-support for searching source code is especially important when the code base is huge or when developers are unfamiliar with it.

The research on code querying has produced many powerful code search languages and frameworks (e.g [2], [3], [4]), that require developers to use a whole new query language with a steep learning curve. On the industry side, the most sophisticated search mechanisms offered by modern integrated development environments (IDEs) are based on graphical widgets. For instance, the Eclipse Java Search Widget has a total of 31 check-boxes and many of them interplay with each other in an unintuitive manner. We'll show later in this paper that users generally don't appreciate this widget (see V). The free text search (like grep) is easy to use, but yields many wrong results (for instance grepping "balance" for the previous example would also list method "checkBalance").

In the same line of thought as naturalistic programming [5], [6], we assume that the most natural way for programmers to express code queries is using natural language. We aim at reconciling a lightweight way to express queries over source code ("à la grep") with the correctness and efficiency of dedicated code query engines. This paper is a novel contribution in this direction.

Development Tasks	Example Query
Change Analysis	Where is the field balance read?
Feature Identification	Where are the classes named *Account*
Bug Localization	Where is an IncorrectRequestException thrown?
Dependency Analysis	Which classes import javax.swing.JButton?

TABLE I
Examples of Software Development Tasks and Related Source Code Queries

In this paper, we present an approach that enables the developer to query source code with natural language. Contrary to [7], 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. It is designed to answer a large range of questions from "*Where is method doSomething called?*" to more complicated queries like "*Where are super reference calls to method doSomething?*". Our prototype implementation is built on top of the Eclipse JDT code query engine [8] 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.

This technical report is a long version of a paper published in the Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering [9], it is organized as follows. Section II gives an overview of our approach. Section III gives a detailed explanation of the system algorithms. A prototype implementation of our approach is presented in section IV. The evaluation of the approach is described in section V. We then discuss related work in section VI and Section VII concludes the paper.

II. Overview

Figure 1 outlines the different phases of our approach to querying source code with natural language. First, the developer has a development task that requires to examine

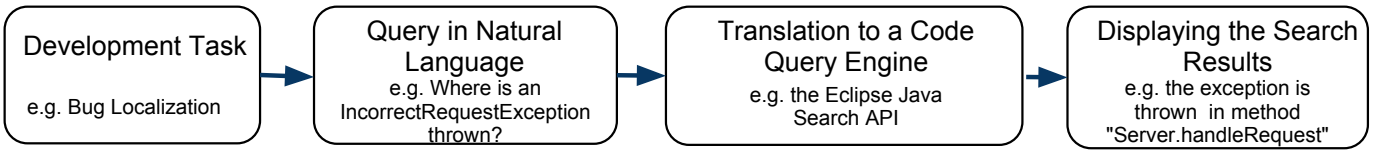

Fig. 1. Overview of our Approach to Querying Source Code With Natural Language

existing source code elements. Then, (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, e.g. the Eclipse JDT Search API. 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 is a bridge between a natural language query and an underlying code query engine. In our work, we have used the JDT query engine. For the sake of giving concrete examples when presenting the approach, we present this engine here, but it is to be noted that our approach is, to a certain extent, independent from the underlying query engine.

The core of the JDT query engine consists 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*”)

To execute a search we have to extract the three parameters from a user’s query. Let us consider the example query *“Where is method splitString called?”*

The first parameter is what kind of source code element the user is looking for. In the example query above, this would be `METHOD`. Some other possible values for this parameter are `TYPE`, `FIELD` or `CONSTRUCTOR`.

The second parameter is the context in which the source code element should be sought. For example, if the user is looking for methods that return an Integer value, just looking for all method signatures containing Integer may yield a lot of incorrect results. By using the context parameter `RETURN`, the search results are limited to methods the return value is of type “Integer”. The context parameter values depend on the element kinds. For instance, `THROWN` is a context for `EXCEPTION` and `FIELD_ACCESS` is a context for `FIELD`. However, there are some generic context like `DECLARATION`, corresponding to where a source code element is declared (as opposed to used). In the example query above, the context is `REFERENCE`, meaning method calls when one searches for methods.

The third parameter identifies what the user is looking for. In the query given above, this is `splitString`. The expression may contain wildcard characters to express a family of identifiers. The supported wildcards are * for any number of characters and ? for single characters. Hence, `split*` refers to all methods starting with `split`.

Eventually, for the example query above, the corresponding API call would be `search(METHOD, REFERENCE, "splitString")`.

III. From Natural Language to Formalized Queries

Figure 1 outlines the different phases of our approach to querying source code with natural language. 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 2, 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 III-E).

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*").

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 [10]. 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.

1) 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.

Training data can also use different query words to refer to the same information. For instance, in "*where are this:context reference:context calls to playit:expression*" the first determiner ("this") and the first noun ("reference") together form a tuple indicating the context parameter, we call this a *tuple-annotation*. The second noun (playit) is the expression parameter value. Furthermore, it is not necessary for a training query to contain values for all search API parameters. The previous training query lacks data for the kind of source code element parameter.

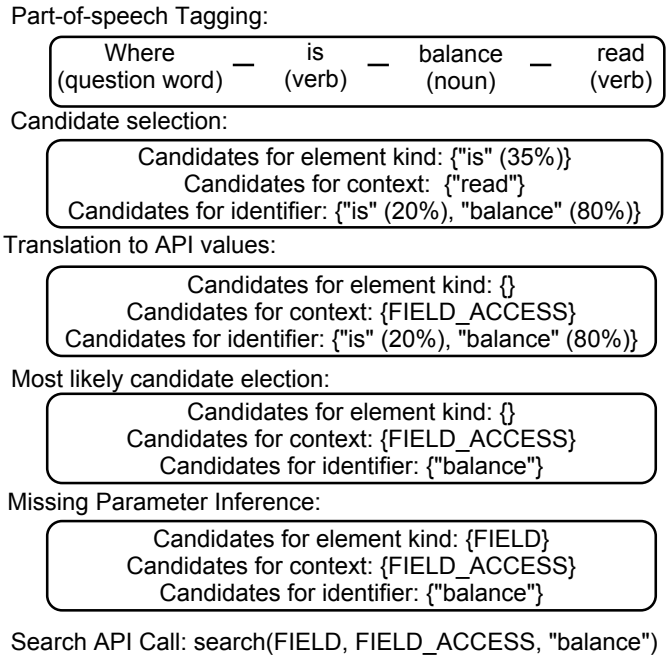

Fig. 2. Example Sequence of the Transformation of a Natural Language Query to an API Call with Valid Values.

Annotated queries are defined based on a combination of the implementor's expertise and real-user data collection. For replication, ours are published on [11]. 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.

2) 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 III-B) 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.

Note that if an annotated query contains tuple-annotations, e.g. "*where are this:context reference:context calls to playit:expression*" which contains twice the annotation "context", then a tuple a containing the corresponding values is added to the candidate list (and not two separate words). In the previous example, the tuple <first determiner, first noun> (with the real values from the query) is added to the context candidate list.

The important part is that the candidates do not contain real words like for example *type* or *method* but only references to specific POS tags like for example the first noun.

3) Performance Consideration: This candidate selection mechanism has different key characteristics that allow optimizing the implementation: 1) computing the grammatical form of each training data can be done offline in order to save POS-tagging time. 2) for a given grammatical form, the probabilities can also be computed offline and stored. Doing so, when a users enters a query, the system only has to find the matching grammatical form. 3) comparing two grammatical forms is comparing two ordered lists, this can be done in better than O(n) where n is the number of training queries.

D. 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").

In order to give the user the freedom to enter queries in a number of different syntactical styles (e.g. past or present), we first stem all candidate words (except for the identifier candidate list, because the corresponding API parameter is free text). Stemming consists of transforming words to their morphological root form. For instance, the words *type* and *types* are both stemmed into *type* and the words *call*, *calling*, *called* are stemmed into *call*.

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.

The target domain (the values of the mapping data) depends on the search API used. For instance, the JDT Code Search API contains two enumeration parameters: the kind of source code element and the context. Since it has 11 possible kind of code element parameter values and 27 possible context parameter values, the target domain contains at most 11 + 27 entries. The input (the keys) domain is made up of words that are commonly used to refer to the respective programming language concept (e.g. class, method, read, super, etc.). In our prototype, the mapping for element kinds contains 12 entries, the mapping for the context contains 46 entries. For replication, our mapping data is published publicly available [11].

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".

E. 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

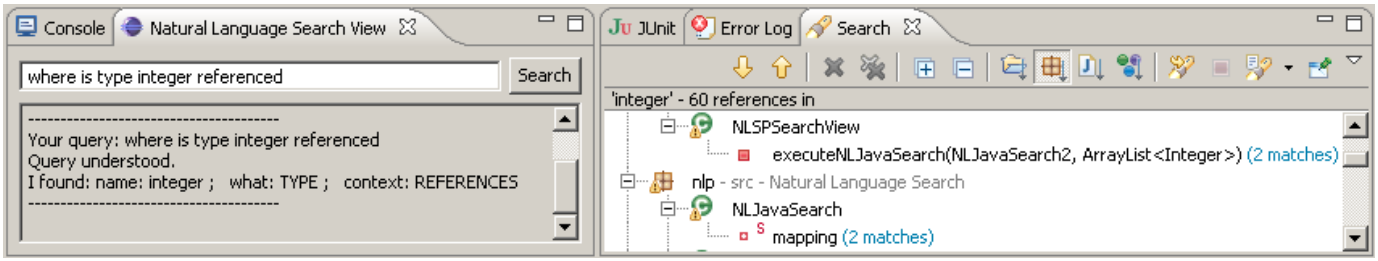

Fig. 3. Screenshot of our Prototype Implementation in the Eclipse IDE. The graphical interface contains a query field, a feedback area, and a search result area.

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 III-C2). 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 III-E1.

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).

1) 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.

Inference rule #2: If we have no annotated training query corresponding to the query's grammatical form, the following fallback strategy is applied. For the context and kind of source code element parameters, we search the query for words which are contained in the key set of the mapping (in stemmed form). If we find one, its translation is used as the respective parameter value. If more than one word of the query for one parameter is contained in the respective key set, the first one is taken. If no word could be found, the parameter value is set to unknown.

2) Performing the Code Search: Figure 2 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 3.

IV. Implementation

We implemented a prototype of our approach for searching Java source code. It is implemented using the Eclipse JDT Search API, which can be used to carry out all kinds of search tasks on java source code. Our prototype is integrated into the Eclipse IDE as a plug-in. A screenshot is shown in figure 3.

A. Graphical User Interface

The user-interface consists of a query field (top left hand side of figure 3), a feedback area (bottom left hand side of figure 3) and a search result area (right hand side of figure 3). The search result area is a standard Eclipse component and the user can directly jump to the respective code by clicking on the search results.

The feedback area tells the user whether the system has understood the query. For example, if the algorithm could not find a value for the kind of code element parameter, it will show the following error message to the user: "*Sorry, I didn't completely understand your question. Please also specify what you are looking for. (eg type, method, etc)*".

For each query, the feedback area also shows how the tool interpreted the developer's query by showing the values for each of the code query parameters, or unknown in case it did not find a value. This is useful to trust the system, as well as to get hints on how to reformulate the query when it is not understood.

B. Query Preparation

We use the LingPipe [12] natural language processing library for POS tagging. For stemming words of the query to match them with mapping keys we use the porter stemming algorithm [13].

C. Default Parameters

The JDT Search API needs four parameters to carry out a search. These are search string (corresponds to the identifier), search for (corresponds to the kind of source code element) and limit to (corresponds to context). The fourth parameter describes where to search, for example, the current project or the whole workspace. It also describes which kinds of sources to include, for example java source files, jar files or the java runtime environment (JRE) libraries. Our prototype defaults this parameter to the current project with only the java source files included.

V. Evaluation

A system to query source code with natural language is useful if it correctly understands a wide range of user queries. To evaluate this, we have created a suite of automated test queries that check the ability of our prototype to translate common queries into correct parameter values of the underlying code query engine (see V-A). However, the major threat to validity of this test suite is that it has been done by ourselves. Even if we took special care to envision all possible ways to ask queries, we surely have missed a lot of different query forms since we can not imagine all possible ways of asking queries.

Consequently, we set up an experiment to collect queries from developers. This experiment enables us to collect queries from people with different programming background and level, native language, and personal ways of expressing queries. The following presents first the automated test queries and then the experiment and its results.

A. Automated Tests

For each of the 37 kinds of query support by the JDT query engine in our prototype, we manually defined 201 test queries (between 3 and 9 unit test cases per kind of query). Those tests were used to design the system as well as to check for regression. They represent different ways to formulate a query. They also include versions of queries using wildcards and with and without marked search strings (string expressions). The unit tests check whether the query is translated to the correct search API parameter values. For instance, the following test queries are used to test the translation algorithm for the following parameter value combination: *type* as element kind, *catch clauses* as context and *NullPointerException* as identifier.

- What catches exceptions of type "NullPointerException"?
- What catches exceptions of type NullPointerException?
- Where are NullPointerException catched?
- Where is a NullPointerException catched?
- Where are catch clauses for NullPointerException?
- What catches NullPointerException?
- Where are NullPointer* catched?

Note that the last test query checks whether the identifier equals *NullPointer** because the wildcard resolution is done by the search API and not the translation algorithm.

The test queries are complete in the sense that they contain all the necessary information to translate them to valid API parameters. Furthermore, all grammatical forms are already in the annotated training data. Note that the fact that test queries are also contained in the training data does not influence the results of the unit test, since the actual words of the training queries are irrelevant and only the syntactical structure is used in the translation process.

We designed and trained our prototype so that it understands 100% of the test queries. For replication, the test suite is publicly available [11].

B. Experiment

The goal of the experiment is to evaluate whether our prototype systems understands queries from real users working on a unknown Java project. Interesting metrics are how many queries are understood or how long it takes the subjects to find the information they are looking for. The experiment also enables us to find out the main weaknesses of the current approach and prototype. The experiment is not a controlled experiment (e.g. comparing subjects with the natural language search tool and subjects with the search widget of Eclipse). The main reason comes from the difficulty of recruiting subjects: on the one hand we would like to collect the maximum number of queries, on the other hand, meaningful statistical tests for controlled experiments require a large number of subjects[1].

1) Experiment Tasks: The subject is given an open development environment (Eclipse v3.6) and 13 tasks. The tasks consists of working on a Java project implementing the game "Space invaders"[2]. The source code of the Java project consists of 13 classes, and a total of about 2400 lines of code[3]. The project is unknown to the subject, and they do not get any information about it. The tasks that the subject must complete are:

1) Method init() is called in a method where it doesn't make any sense. Remove the method call.
2) Field channelNum is wrongly set to 5. Fix this by setting it to 1.
3) One of the types in the project extends type BaseClass. This isn't necessary anymore since BaseClass will be removed from the project. Change it accordingly.
4) All types that implement interface IRenderable should also implement ICollidable. Check the code and change it in case it has been forgotten somewhere.
5) Remove unused method playIt().
6) Add a System.out.println("Instance created") right after an instance of SoundManager is created.

[1] e.g. 52 subjects (26 subjects for each group) for an independent sample t-test at 95%/5% with estimated $\mu_1 = 15min$, $\mu_2 = 10min$, and $\sigma = 5min$

[2] which is included as an example in the Lightweight Java Game Library, seehttp://www.lwjgl.org/

[3] For replication, the project is available at [11]

7) Add a System.out.println("Possible source of NullPointerException"); to every method that throws a NullPointerException.
8) If exceptions of type NullPointerException are catched, a text message stating this fact should be printed to the console.
9) Print the value of field lastId right after it is read.
10) The import of class DropTarget isn't necessary. Remove it.
11) Rename field numbe to number. numbe is a typo.
12) There is one method in the project that takes an Integer argument. Print the value of the argument by adding a System.out.println to the method for debugging reasons.
13) Remove all Deprecated annotations. It has been decided that the annotated code will be used in future releases.

2) Development Environment: 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.

The code snippets to change are distributed over all classes on the project to avoid the situation where the subject already knows where to find the code for a task because he accidentally found it while working on an earlier task. The names of the objects and methods mentioned in the tasks give no hint of where the sought code is.

3) Experiment Setup: 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 experiments concludes with an oral and recorded interview structured around 4 topics (see V-B5).

Subjects are given a 5 minutes oral presentation on how to enter queries, what information should be included in the queries, where the results are shown and how to use them, what kind of error messages the tool displays, what to do if the tool does not show any result or wrong result and any special features like the possibility (not required) of marking search strings with double quotes to help the system. Note that the subject does not know the final goal of the experiment (collecting queries) so as not to interfere with the way of expressing queries. Also, since our prototype is novel and not publicly available, we are sure that the subjects have no experience with the tool.

Any user interaction with the natural language search tool is logged. The logging information includes the query entered by the user, the response of the tool (like for example *Query understood*), the translation information of the tool (the exact values it found for the three parameters of the JDT query engine) and a time stamp for each entered query.

4) Pre-Experiment and Final Experiment: We run two versions of the experiment: a pre-experiment and a final experiment. The pre-experiment has three goals. First, we wanted to have a first impression on how the prototype performs. Second, we aimed to collect data to improve the prototype (training data, mapping data). Third, it was also a way to get feedback on the experiment itself and to validate the experimental protocol. Two months before the main experiment described here, the pre-experiment took place with 6 subjects, not compensated. It fulfilled our expectations and the setup remained the same for the main experiment, except minor changes to the tasks like field name changes and small reformulations.

The main experiment was carried out with an enhanced version of the prototype based on the collected data and observations during the pre-experiment. For instance, we added data to the training queries and mapping information. We also refined parts of our approach and algorithms.

We recruited 14 people for the main experiment, all compensated. 11 of them were students and 3 of them were graduated computer scientists. We recruited 3 by using internet forums or mail invitations, the rest were directly recruited with on-place contact at the university where the experiment was held.

All numbers and analyses given in the following sections are derived from the main experiment (14 subjects), since it uses the latest and best version of the prototype. The most important data, the number of understood and correctly understood queries for the pre-experiment is given in table III.

5) Interview: The last 5 to 10 minutes of the experiment are used to interview the subject about his/her background and the natural language search tool. This interview was only done with the subjects in the main experiment. First, the subjects were asked how they would assess their Java programming skills on a scale from 1 to 5 (with 1 being beginner and 5 being expert) and their native language. This is used to classify the experiment results.

The second set of questions is about the qualitative evaluation of our prototype. The subjects were asked about their general impression of the prototype, what they liked about the idea and whether they would use the tool in their daily work. We also asked questions about the standard Java search widget offered by Eclipse (whether they know it and like it). Also, we asked whether they would prefer the default Eclipse search widget or the natural language search tool if given the choice between the two. The reason for this is that the widget uses the same API as our prototype implementation (i.e. they have the same expressive power), but the look'n'feel of using it is very much different (Eclipse search uses a standard graphical user interface using radio buttons and text fields). The qualitative analysis derived from these interviews is given in section V-B9.

6) Quantitative Results: With the data acquired through logging, we derived a number of statistics to analyze how well our approach works under our experimental conditions.

The most important ones are the number of understood queries and the number of correctly understood queries. These have to be separated, because the fact that the tool understood

Characteristics	Value
Number of programming tasks	13
Number of subjects	14
Total completion time - Average	9 min
Total completion time - 90% percentile	[6,10] min
Total completion time - Min	6 min
Total completion time - Max	16 min
Task completion time - Average	41.5 sec
Task completion time - 90% percentile	[10,60] sec
Task completion time - Min	10 sec
Task completion time - Max	2.5 min
Average # of tasks solved in one query	9

TABLE II
DESCRIPTIVE STATISTICS OF THE EXPERIMENT

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.

The average number of queries the subjects needed to solve all 13 tasks was 20. This includes queries which were not understood and wrongly understood. It took them an average of 17 correctly understood queries to solve the 13 tasks.

It took the subjects an average of 9 minutes to finish all tasks or an average of about 41.5 seconds to finish one task (reading the task, finding the correct query, checking that the found code is correct).

In average over all subjects, 9/13 tasks were be completed with a single query (i.e. the first query was correctly understood). The four tasks that the subjects could not solve with one query were different for all subjects. There was no task that no user could solve with one query, nor there was a single task that every user could solve with one query. The lowest number of tasks a subject was able to complete with one query was 8. Also, one subject was able to solve all 13 tasks with 13 queries only. We observed that most users had one or two tasks for which they needed a lot of queries, up to 9 for a single task for one subject.

The average Java proficiency score the users gave themselves was 3.15 out of 5 with the highest score being 4 and the lowest being 1 to 2 (counted as 1.5). Despite the difference between the highest and the lowest score, we could not find any patterns in the amount of understood and correctly understood queries related to Java skills of the subjects.

In terms on native language, 9 subjects were German, 1 Chinese, 2 Vietnamese and 1 Russian. We also found no relation between query understanding ratio and native language of the subjects. We figured that the time it took the German subjects

Input Queries	# Queries	% rec	% correct
Test suites	201	201 (100%)	188 (94%)
Pre-experiment	146	114 (78%)	91 (62%)
Final experiment	276	251 (91%)	228 (83%)

TABLE III
PERCENTAGE OF UNDERSTOOD AND CORRECTLY UNDERSTOOD QUERIES FOR THE TEST SUITE, THE PRE-EXPERIMENT AND THE FINAL EXPERIMENT

to phrase their queries was an average of 40% lower than the time it took the other subjects (chinese, vietnamese and russian). This may be caused by the lower language distance between english and german compared to chinese, vietnamese and russian. We also found that the graduated subjects (3/14) had higher average recognition rates, 94% of their queries were understood. We assume that they are both more comfortable with English and with Java, resulting in clearer queries.

7) Analysis: Not/wrongly Understood Queries: In the following, we analyze the reasons behind the 17% of queries that were not or incorrectly understood.

The main reason is missing information. For example, if the subject enters the query "*Where is init?*" the tool has no information at all to deduce what the subject is looking for (e.g. method or class) This problem accounted for 51% of the queries that were not understood.

Another reason is the use of an word that is missing in the mapping dictionary. For example, a subject working on task #9 entered the query "*Where is field lastid used*", which was not understood, even if the necessary information is present. The reason is that the mapping dictionary does not contain "use" (the stemmed root of used) as a key. Unknown keywords accounted for for 35% of all queries which were not understood. Note that since that unknown keywords can be easily added to the prototype configuration, those queries would be understood by simply feeding the prototype with new mapping entries.

It also happened that the subject entered a query which contains all necessary information (including known keywords), but misspelled them. Taking again task #9 as an example, a subject entered the query "*Where is field lastid red*" which was not correctly understood. The reason is that the subject misspelled "read" in "red" and thus no mapping entry was found. This problem is responsible for about 15% of all queries that were not understood. Note that sometimes a query presented different symptoms hence the sum of percentages is higher than 100%.

Let us now consider the queries that were incorrectly understood. After analysis, we figured out that in the candidate election phase (see III-C2) the candidate with the highest probability was the wrong one. This problem could be mitigated by enriching the training data of the system (see III-C1).

8) Analysis: Correctly Understood Queries with No/Wrong Results: Not all correctly understood queries yield search results usable to solve a task. Some even yield no search results at all. There are two reasons for this.

The first reason is that the subject did not enter a query expressing what he was actually searching for. For example, one subject entered the query "*Where is method Nullpointerexception*" for task #7, so the tool obviously retrieved unwanted information. This accounted for about 75% of all correctly understood queries which did not yield the result the subject was looking for. We assume that this is due to subjects being not completely proficient with the vocabulary and concepts of the programming language.

The second reason is that the subject made a spelling error. For example, one subject entered the query "*Where is field **lasid** read?*" for task "*Print the value of field **lastId** right after it is read.*" (task #9). The query was correctly translated to the code query engine, but the search string *lasid* does not yield any search results because there is no field with this name. This accounted for about 25% of all correctly understood queries which did not yield the result the subject was looking for.

Interestingly, we noticed that the subjects who did not find an answer with the first query usually took more than 2 tries to do so. We explain this phenomenon as follows. The main reason is that many subjects seemed to ignore error messages (according to our visual observations during the experiment). For example, a subject entered the query "*Where is parse referenced*" and the tool complained about missing information on the element kind. The tool also displayed a hint ("Are you looking for type or method?"). However, the subject (and other subjects in similar situations), ignored the message and changed something else in the query. For example instead of just adding type or method to the query, they for example changed it to "*Where is parse defined*", and started another search. Also, it happened that the subjects forget what they had already entered and repeated an incorrect query that they already asked.

9) Qualitative Analysis: In this section, we present the results derived from the second part of the interview as described in section V-B5.

The first observation we made was that all subjects had an overall positive impression of the tool. No one said he did not like the idea of code search with natural language queries. On the question what they liked, the most important point was that 7/14 subjects thought the tool was flexible, both in the number of different queries it could understand as well as the number of different ways the respective queries could be phrased.

On the question what they did not like, 3/14 subjects pointed out that they thought it was hard to think of a query the tool would understand and 4/14 said the tool needs to be more flexible, meaning it needs to be able to understand more different queries. 5/14 subjects said the the user interface of the tool needs to be improved, for example by making the input line accessible through a keyboard shortcut and improving the presentation of the feedback information. 4 subjects did not have any negative points.

On the question whether they would use the tool in their daily programming work, the results were split. 8 subjects said that they would use the tool in big projects with lots of code, projects which are unknown to them or for more complex queries like for example finding all catches of certain exceptions. 4 subjects said they would also use the tool for smaller projects or simpler tasks like finding references to a type or a method declaration. The reason was that they think that writing a natural language query for this is takes too long and that for simple queries they feel good with existing IDE tools they know and use (mainly the outline view or find references shortcut).

Experiment subject opinion	**%**
I generally like the idea	14/14
I think it is flexible	7/14
I would use NLPSearch for my programming duties (for small projects)	4/14
I would use NLPSearch for my programming duties (for big projects)	8/14
I use the Eclipse Search Widget	5/14
I prefer NLPSearch over the Eclipse Search Widget	4/5
I think it should understand more forms of queries, it should be more flexible	4/14
I had to think to find correct queries, it was not intuitive	3/14
I think the UI could be improved	5/14

TABLE IV
MAIN POINTS GATHERED FROM SUBJECT INTERVIEWS

We questioned the subjects about the standard Eclipse Java search widget, which offers the same functionality as the natural language search tool but with a standard graphical user-interface. 9/14 of the subjects said that they did not know it. Of the 5 subjects that knew the widget, 3 said they use it regularly. We also asked whether they would prefer the search widget that Eclipse offers or the natural language search tool. 4/5 subjects familiar with Eclipse Search said they would prefer the natural language interface, because they think it is more intuitive and easy to use.

Table IV gives an overview of these results. Overall, 7/14 subjects think that the system is not flexible/intuitive enough. This percentage has to be put in perspective together with the percentage of correctly understood queries (83%). This shows that this lack of flexibility is not an incapacity to understand queries. Those 7 subjects could not express queries in their most natural and intuitive way, they had to perform a mental translation before expressing a query. However, 3 subjects complaining about flexibility would still use our tool for big projects or prefer it over the Eclipse search widget. This shows that that our approach reduces the translation gap between the mental state and the actual interaction with the code query engine.

C. Performance

We also measured the performance of the prototype implementation to see whether it is fast enough to be used in a real work environment, where it is crucial the the search results come up as fast as possible. We have measured the translation time for all queries of the test suite. The result us that translating a natural language query to concrete search API parameters takes between 1 and 2 milliseconds. Also,

during the experiment, no user complained about the response time.

VI. 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., [4], [14]. 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 [15] described different software maintenance queries. Hill et al. [16] uses NLP based on program identifiers to improve contextual code search (which pieces of code are about this topic?). Ko et al. [17] presents a system for querying program output, and not source code itself. Wang et al.'s approach [18] detects duplicate bug reports using NLP.

There is also an interesting research thread on natural language interfaces for databases. A excellent overview of this field is given by Androutsopoulos et al. [19], an example system is Query Builder [20]. The main difference with our work is that we consider source code as "data". However, if one feeds a database having a code oriented data model with source code, the approaches become comparable. This requires a heavyweight infrastructure and poses a number of performance issues. On the contrary, our approach translates queries to an off-the-shelf code search component (the Eclipse JDT query engine), that is optimized, mature and used in large-scale projects.

The notion of "conceptual queries" for software development has been discussed by de Alwis and Murphy [15]. The system they describe supports a fixed number (36) of queries. On the contrary, our system enables a wide range of queries (supported by the underlying code query engine), and especially a large number of variations in the manner to formulate queries.

Würsch et al. [7] 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.

VII. 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.

Future work is guided by the feedback collected during the user-study. First, we will integrate a system to handle typos, which severely hinder the performance of the whole system. Second, it has been recognized that the user-interface can be much improved, in terms of interaction (keyboard only must be supported too) and in terms of feedback (to provide more guidance when the systems fails to correctly understand a query). Third, a query completion system would be appreciated, especially since users are used to interact with code completion systems and with search engine suggestions. Finally, we plan to adapt the approach to more powerful query engines, in order to support more specific queries, such as "Where are synchronized blocks?". This will also be the occasion to have a better view on the overall genericity of our translation scheme.

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] S. Paul and A. Prakash, "Querying source code using an algebraic query language," in *Proceedings of the International Conference on Software Maintenance*, pp. 127–136, IEEE, 1996.

[3] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, pp. 463–475, 1994.

[4] 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.

[5] C. Lopes, P. Dourish, D. Lorenz, and K. Lieberherr, "Beyond aop: toward naturalistic programming," *ACM SIGPLAN Notices*, vol. 38, no. 12, pp. 34–43, 2003.

[6] R. Knöll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 542–559, ACM, 2006.

[7] 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.

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

[9] M. Kimmig, M. Monperrus, and M. Mezini, "Querying Source Code with Natural Language," in *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering*, pp. 376–379, 2011.

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

[11] M. Kimmig, M. Monperrus, and M. Mezini, "Replication data." http://www.monperrus.net/martin/nlsearch, 2011. Accessed December 2014.

[12] Alias-i, "Lingpipe 4.0.1." http://alias-i.com/lingpipe, 2008. Accessed March 1, 2011.

[13] M. Porter, "The porter stemming algorithm." http://tartarus.org/~martin/PorterStemmer/, 2008. Accessed March 1,2011.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases - an introduction," in *Natural Language Engineering*, pp. 29–81, 1995.

[20] T. O. J. Little, M. de Ga and R. Alhajj, "Query builder: A natural language interface for structured databases," in *Computer and Information Sciences - ISCIS 2004*, Lecture Notes in Computer Science, pp. 470–479, Springer Berlin / Heidelberg, 2004.