Abstract—To support programmers during program maintenance we present an approach which extracts concepts and relations from the source code. Our approach applies natural language parsing to sentences constructed from the terms that appear in program element identifiers. The result of parsing can be represented as a dependency tree. Then, we automatically extract an ontology by mapping linguistic entities (nodes and relations between nodes in the dependency tree) to concepts and relations among concepts. We applied our approach to a case study and assessed the result in terms of the support it can give to concept location, executed in the context of bug fixing.

I. INTRODUCTION

Program understanding involves rebuilding the “mental model” of the domain knowledge captured in a program, which requires identifying the domain concepts, and the relations among them. Usually, programmers learn about these concepts from program element names, which approximately constitute 70% of the source code [1]. Realizing this, different approaches and conventions have been proposed to analyze and improve program element names [1]–[4]. However, despite the efforts, rebuilding the “mental model” of domain knowledge is difficult. This is due to the different perception and representation of domain concepts in the program by different programmers at different times during the program’s evolution.

We propose the use of natural language parsing (NLP) techniques to extract domain concepts and relations from program identifiers. To achieve this, sentences are constructed from list of terms that compose an identifier and parsed using NLP. The linguistic dependencies produced after parsing are then used to recognize concepts and relationships that form our automatically extracted ontology.

To evaluate our approach, we have conducted a case study in which the concepts available in the ontology have been navigated so as to refine the queries used in the execution of a concept location task. The results obtained from the case study are encouraging and indicate that ontology concepts can contribute to a substantial reduction of the concept location effort, thanks to the improved precision of the queries which include ontology concepts.

II. CONCEPT EXTRACTION

Object oriented programmers usually follow widely known and commonly adopted naming conventions. They use the same (often implicit) grammatical rules for names of program elements which are structurally the same. These rules tell the role of each term in a name. For example, method names are often constructed from verbs which are followed by nouns, while class names are frequently sequences of nouns. The roles of the terms can be exploited to mine for concepts and relations among them. This can be achieved by parsing the names using natural language parsers like Minipar which identifies the lexical category of each term and the natural language dependency among them.

The steps followed by our approach are explained below. To demonstrate the process, we use the example in Figure 1.

1. Creating term list: The list of terms is created by splitting class, attribute and method program element names into their composing terms. For splitting, we take advantage of commonly used ways to separate terms in program element names, such as camel casing (e.g., processMail) and underscore. When the terms resulting from splitting are known abbreviations or contractions, they are automatically expanded [5]–[7].

2. Generating candidate sentences: Minipar takes sentences as input for parsing. However, the term lists created in step 1 are not complete sentences. Hence, we have formulated different rules to construct sentences from the list of class, attribute and method terms (see Table I). For each term list, a rule is applied when the corresponding constraint is satisfied. To see if a constraint is satisfied (i.e. if the first term in a list can be used as a verb or noun), we have used WordNet. For example, for the term list \( \text{process, mail} \) two candidate sentences are generated by applying both MR3 and MR4: “\text{subjects process mail}” and “\text{subjects get process mail}”, since the term \text{process} is both a verb and a noun according to WordNet. If none of the constraints is satisfied for the given list of terms, the list is used as is.

3. Sentence selection: Applying the rules in Table I gives a maximum of two candidate sentences for a term list. Hence, we need to select a sentence among the two for the next step, concept extraction. Prerequisite to the selection is NLP...
of the candidate sentences. Minipar can be used for this purpose. Once parse trees are available, we apply the following selection criteria in the following order:

a. If only one of the parsed sentences have a U in the result, select the one without U. When Minipar is not able to identify a term in a sentence, U(Unknown) is reported.

b. If both sentences do not have a U and the source of the terms is a method, the method name is checked against the attributes of the enclosing class. If a match is found, the sentence with the verb get is selected.

c. If both sentences do not have a U, select a sentence based on the frequency of the role of the first term of the list (e.g., verb and noun) in the sentence. The highest frequency role is selected, with the frequency of each role obtained from WordNet.

d. If both sentences do not have a U, the sentence with user defined higher priority is selected. If the sentence was generated for a method, the verb role is given higher priority. Otherwise, the noun role is preferred.

e. If both sentences have a U, apply selection criterion d.

For example, method read from the running example generates two candidate sentences: $S_1 = \text{"subjects read object"}$ and $S_2 = \text{"subjects get read"}$. These two sentences are parsed correctly (with no U in the parse tree). The term read does not appear in the class attribute names. According to WordNet, the frequency of use of the term read as a verb is much higher than the frequency of the noun. Hence, based on criterion 3 $S_1$ is selected for further analysis.

4. Extracting concepts and relations: The concepts which are used in building the ontology are derived from the nouns in the term lists. The ontological relations are obtained by mapping the linguistic relations in the dependency tree produced by Minipar to ontological relations. The linguistic relations of interest are $obj$ which is a natural language dependency relationship between a verb and a noun that plays the role of object, and $NN$ and $mod$ which are natural language dependency relationships between nouns or adjectives and nouns.

The target ontological relations and the corresponding natural language dependency relations to which they are mapped are described below.

- $isa$: used to connect general and more specific concepts.
  
  It is derived from $NN$ and $mod$ linguistic relations.

- $<verb>$: a context specific relation between a concept and the object on which the verb acts. The type of relation identified between the concept and the object is taken from the term that plays the role of a verb which, in this case, is a non-accessor verb. The linguistic relation, which corresponds to it, is $obj$. If the list of terms contains only one verb, the $<verb>$ relation is between the program and class name. While, when the list contains a verb and an object, the relation is between the class name and the object.

- $hasProperty$: is a relationship between a concept and its properties. It is derived from the linguistic dependency relation $obj$ between an accessor verb and an object.

In the sentence “subjects add panel field” constructed from our running example (Figure 1), two concepts, panel field and field, are generated and the NN natural language relation between panel and field is mapped to an isa relation in the ontology, originating isa(Panel field, Field) (see Figure 2). The object in the sentence is associated with the concept Panel field. The enclosing class for the method addPanelField is associated with the concept Merge gui. Hence, a context-specific relation add(Merge gui, Panel field) is created in the extracted ontology.

### III. Case study

To assess the support our approach provides to programmers, we have conducted a case study in the context of concept supersisation location. In this regard, we have formulated two research questions:

**RQ1:** Do the extracted ontology concepts contribute to increasing the precision of programmer's queries formulated for concept location?

1. Ontology concepts should not be confused with programmers' concepts to be located in the code. We qualify the first term as ontology concept to disambiguate when necessary.
RQ2: Do the extracted ontology concepts contribute to increasing the recall of programmer’s queries formulated for concept location?

In the context of concept location, where programmers query the code base and investigate the result to determine the exact location of a change, Precision (P) is the ratio between the number of files correctly identified and the total number of files retrieved as the result of a query. Low precision indicates that programmers have to investigate many files to identify those relevant to their task.

Recall (R) is the ratio between the number of correctly identified files retrieved using a query and the total number of files to be actually modified (i.e., the files that should be retrieved by the query). It describes the completeness of the files retrieved to address the bug or change request.

Usually, precision and recall are inversely related and hence difficult to compare with results of other queries. In such circumstances, the F-measure (F), which is computed as $2*(P*R)/(P+R)$, is used to aggregate both measures into a single value. In this case study, we investigate whether using the extracted concepts to formulate a query and retrieve files improves the precision and recall, and ultimately the F-measure.

The subject of our case study is WinMerge2 version 2.12.2 which is a medium size program written in C++. We have analyzed 257 files (67,637 lines of code; 138 C++ classes) which implement the core functionality. WinMerge has a bug tracking system and dedicated website on which the developers provide information about the releases and major bug fixes incorporated in each specific release.

A. Experimental procedure

To conduct our case study we have selected seven bugs among the major bugs reported on WinMerge’s website. These bugs are closed and have patches. From the bug descriptions and patches we have manually gathered keywords and names of actually modified files, respectively. The keywords are terms that appear in the bug descriptions and that we think represent a specific concept related to the bug reported. The file names are used as gold standard for evaluating the results of queries.

We have developed a tool that implements the steps described in Section II to extract concepts and build the ontology. The concepts selected to enhance a query are obtained by first locating the keywords in the ontology and then collecting the neighborhood concepts, i.e. those concepts which are a single edge away from the keyword concept. To visualize the ontology and determine neighboring concepts, we have used Protege3.

A developer may either select the most relevant keywords from the bug description to formulate the query (basic query), or s/he can select the most relevant ontology concepts to be included in the query (enhanced query) from the neighboring concepts of the keywords. The selection is done manually, based on the subjectively assessed relevance of the neighboring concepts for the bug description being considered. When the number of files retrieved by the query is too high, the developer may use other keywords or concepts to filter and narrow down the result.

B. Results

Table II provides a summary of our case study. It shows the keywords extracted from the bug description and the corresponding concepts collected from the ontology. In addition, it shows the basic and enhanced queries executed, together with the related precision, recall and F-measure. The presence of a right arrow (→) between two queries in Table II indicates that the output of the first query was filtered by running the second query on it.

With respect to RQ1, the results show that the extracted concepts have improved the precision of the concept location task by about five times in four out of the seven cases. Indeed, a major precision improvement was obtained (4 vs. 28 files to be inspected, on average), in the cases where ontology concepts have been added to the query. Recall, which is the focus of RQ2, remained unaffected in all the cases but one, in which it was halved. F-measure was always greater or equal for enhanced vs. basic queries.

C. Discussion

The results show that the use of our approach has increased the precision of concept location while reducing the search space. This is obtained at almost zero cost of ontology construction as the process is completely automated. The increased effort in query formulation was negligible. Hence, the approach potentially saves developers a lot of effort and time required for change location tasks.

Most bug reports provide a high level description of the bug, which contains only few keywords that are useful for locating the relevant files. However, this case study demonstrates that, following our approach, the developers can get additional concepts that help them reduce the search space and retrieve the files relevant for fixing the bug (Example: merge edit view for bug 2787131).

In addition, a bug description might not contain the actual terms which are used in the source code to represent a concept. Using the ontology, developers can identify the terms which are actually used in the code, since traceability between ontology concepts and code is granted by construction (Example: desc in bug 2772646).

IV. Related works

Building a relation between the concepts in the programmer’s mind and program components is one of the approaches that facilitates program comprehension and concept location [8], [9]. This can be achieved by mapping (high level) concepts to source code elements using the lexicon they use. In this regard, [10]–[12] have formulated an approach and developed tools that are used to query the source code and locate program components.

2http://www.winmerge.org
3http://protege.stanford.edu
Petrenko et al. [13] have used fragments of ontologies to partially comprehend a program and formulate a query to locate concepts in the source code. Our approach is in line with this work, the main difference being that we use NLP to automatically extract ontology concepts, instead of relying on ontology fragments constructed manually by programmers. Ratiu et al. [4] have developed an approach to automatically extract a domain ontology from different domain specific APIs that target the same domain. Ratiu et al.’s approach uses similarity and path matching to identify concepts from the APIs. In our approach, we identify and extract domain concepts and relations by applying NLP techniques. These techniques allow us to extract domain information from single software programs overcoming the need to find similar APIs.

V. CONCLUSIONS AND FUTURE WORK

We have extracted domain concepts from program element names by applying NLP and organizing such concepts into an ontology. To validate our approach, we have used the concepts in the ontology to (re-)formulate queries used in concept location. The results show that queries including ontology concepts reduce the search space, when determining the files relevant to the change request. Ontology concepts increase the precision of the query in more than 50% of the cases considered, while keeping recall almost always unchanged. On average, the increased precision can be quantified as around 86% less files (4 vs. 28 files, on average) to inspect. It is achieved at negligible extra costs, since the ontology extraction technique is automated.

Currently our approach expects limited use of structural and programming language specific information. For example, it does not handle general utility classes separately from the domain specific ones, and it does not consider structural relationships such as extends and implements. In our future work, we plan to incorporate such information into our approach. Moreover, additional lexical categories, grammatical relations and sentence generation approaches will be explored to identify potential sources of concepts and to enrich the ontological relationships among them. We will also conduct further case studies to determine an optimal combination of natural language and structural heuristics for concept extraction.

REFERENCES


TABLE II

<table>
<thead>
<tr>
<th>Bug id</th>
<th>Keywords</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>202312</td>
<td>EOL, mixed-EOL, open mixed-EOL</td>
<td>side eol, enol, mix eol, recompute eol, left eol, right eol</td>
</tr>
<tr>
<td>278347</td>
<td>HTML report, doctype, file compare</td>
<td>file cmp, multiple file cmp</td>
</tr>
<tr>
<td>278713</td>
<td>Prediffer, select prediffer, plugin</td>
<td>plugin prediff, diff wrapper, plugin manage, info prediff, merge doc, merge edit view, on prediff, automat plugin</td>
</tr>
<tr>
<td>2791402</td>
<td>EOL, whitespace, whitespace char, turn off</td>
<td>side eol, enol, mix eol, recompute eol, left eol, right eol, main frame, view whitespace, item whitespace</td>
</tr>
<tr>
<td>2791506</td>
<td>Generate patch, time stamp, UTF 16, patch</td>
<td>patch tool, main frame, dig patch</td>
</tr>
<tr>
<td>2798345</td>
<td>file filter, file filter pattern</td>
<td>file filter helper, global file filter, merge app, filter</td>
</tr>
<tr>
<td>2772646</td>
<td>left description, right description, description</td>
<td>filter desc, dir doc, col descript, left desc, right desc, desc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bug id</th>
<th>Basic query</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Meas</th>
<th>Enhanced query</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Meas</th>
</tr>
</thead>
<tbody>
<tr>
<td>202312</td>
<td>mixed.*EOL</td>
<td>5/15 0.33</td>
<td>5/6 0.83</td>
<td>0.48</td>
<td>mixed.*EOL</td>
<td>5/15 0.33</td>
<td>5/6 0.83</td>
<td>0.48</td>
</tr>
<tr>
<td>278347</td>
<td>html.*report ⇒ doctype ⇒ file.*compare</td>
<td>(1/1) 1.0</td>
<td>(1/1) 1.0</td>
<td>1</td>
<td>html.*report ⇒ doctype ⇒ file.*compare</td>
<td>(1/1) 1.0</td>
<td>(1/1) 1.0</td>
<td>1</td>
</tr>
<tr>
<td>278713</td>
<td>prediffer ⇒ plugin</td>
<td>2/25 0.08</td>
<td>2/2 1.0</td>
<td>0.15</td>
<td>plugin.*prediffer ⇒ merge.*edit.*view</td>
<td>1/3 0.33</td>
<td>1/2 0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>2791402</td>
<td>whitespace ⇒ EOL</td>
<td>1/26 0.04</td>
<td>(1/1) 1.0</td>
<td>0.07</td>
<td>left.*eol ⇒ view.*whitespace</td>
<td>1/3 0.33</td>
<td>(1/1) 1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>2791506</td>
<td>patch</td>
<td>1/25 0.04</td>
<td>(1/1) 1.0</td>
<td>0.08</td>
<td>patch.*tool</td>
<td>1/5 0.2</td>
<td>1/1 1.0</td>
<td>0.33</td>
</tr>
<tr>
<td>2789345</td>
<td>file.*filter.*pattern</td>
<td>(1/5) 0.33</td>
<td>(1/1) 1.0</td>
<td>0.5</td>
<td>file.*filter.*pattern</td>
<td>(1/5) 0.33</td>
<td>(1/1) 1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>2772646</td>
<td>description</td>
<td>1/35 0.03</td>
<td>(1/1) 1.0</td>
<td>0.06</td>
<td>left.*desc ⇒ dir.*doc ⇒ description</td>
<td>1/6 0.17</td>
<td>(1/1) 1.0</td>
<td>0.29</td>
</tr>
</tbody>
</table>

TABLE II

PRECISION AND RECALL OF QUERIES WITHOUT/WITH ONTOLOGY CONCEPTS