Utilizing Web Search Engines for Program Analysis

Daniel Ratiu
Technische Universität München
Germany
Email: ratiu@in.tum.de

Lars Heinemann
Technische Universität München
Germany
Email: lars.heinemann@in.tum.de

***Abstract*—Programming involves representing domain concepts by using programming abstractions. In object-oriented programs, concepts and relations of the business domain are represented as classes, attributes and methods. However, the concepts and relations that logically belong together are scattered across different modules, interleaved with technical concepts, and distorted due to implementation details. In this paper, we present an automatic method to identify logically related concepts and the relations among them. To achieve this, we systematically transform program identifiers into fragments of natural language sentences and check whether these sentence fragments are meaningful for humans. In order to automatically perform such checks, we use the World Wide Web as a knowledge base that contains a huge number of meaningful texts, and use the Google web search engine to validate the meaningfulness of these sentences. If the search engine returns a sufficient number of hits, we discovered a piece of knowledge in the code. By systematically applying this method, we obtain a condensed form of the knowledge embodied in the program which is an enabler for automatic analyses. We present our experience with several use-cases: (1) assessing the meaningfulness of identifiers, (2) extracting complex concepts from compound identifiers, (3) extracting a meaningful taxonomy from the class hierarchy, and (4) extracting complex conceptual relations from the code. We report on our observations during the analysis of real world Java code, discuss the limitations of our approach and sketch extension possibilities.**

***Keywords*-concept location, program analysis, analysis of identifiers, domain knowledge**

I. Introduction

Programming involves representing concepts of the business domain in the code. Object-oriented programming languages are advocated to better support the implementation of domain concepts by providing adequate abstraction mechanisms [1], [2]. There is a widespread opinion among practitioners that well structured programs should mirror the structure of their business domain and should be easy to read like natural language prose [3], [4], [5], [6].

In practice, the pieces of domain knowledge are usually scattered across several modules—a phenomenon known as delocalization [7]. In the code, the implementation of domain concepts is interleaved with the implementation of technical concepts. [8]. Furthermore, the domain knowledge is distorted by the use of inappropriate programming constructs [9]. These facts make programs like puzzles: reconstructing parts of the domain knowledge implemented in the program involves identifying the different program modules that implement the concepts and how they fit together. Thereby, concept location [10] and concept assignment [11] are central problems for program understanding that are notoriously difficult to automate.

```
public abstract class Rectangle implements Cloneable, Shape {
    public float x, y;
    public float width, height;
    public Object clone() { ... }
    public int hashcode() { ... }
    public String toString() { ... } ...
}
```

Figure 1. Domain concepts are interleaved with technical details

In Figure 1, we present a code fragment taken from the Java standard API that implements a very simple concept, namely a "rectangle". In this fragment, only half of the words used to form identifiers refer to knowledge related to rectangles – namely, "rectangle", "x", "y", "shape", "width" and "height". All other identifiers represent technical concepts related to programming or to Java technologies. In large programs, the situation is much more complex. The distinction between those words contained in an identifier that represent domain concepts and those that represent implementation details is a challenging task that is hard to automate.

In this paper, we propose an automatic approach for identifying logically related concepts as well as the relations among them. Our approach can be described by the following metaphor: Let's imagine that a robot wants to learn general knowledge that is implemented in a program and that is meaningful for humans. Its approach (illustrated in Figure 2) is to learn new facts about the program by "reading aloud" fragments of the source code and waiting for the response from a human audience. The audience decides whether the read utterances make sense or not. For example, the sentence "a rectangle is a shape" is meaningful, while "a rectangle is a clonable" is not. Hopefully, somebody from the audience recognizes the utterances as meaningful and tells this to our robot. Whenever it gets positive feedback from the audience, it identified a knowledge unit (called *"knowledge*

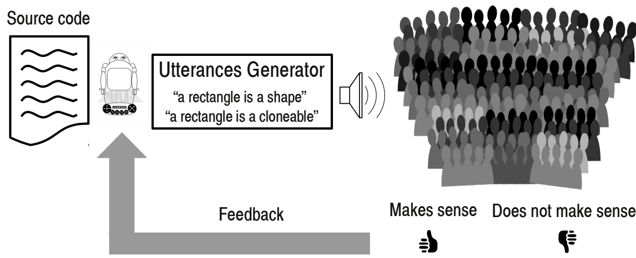

Figure 2. A robot learns new facts about the domain of a program by "reading aloud" the code and obtaining feedback from a human audience

quark"). In order to operationalize this metaphor we have to answer the following two questions:

1) How do we read aloud the program? By reading the program code as it is written (i. e. in a sequential manner), there is a low probability that the human audience recognizes concepts that logically belong together. Therefore, our approach is to maximize the chance that humans would recognize the utterances by transforming the code into (fragments of) English sentences based on carefully chosen heuristics.

2) Where do we find the patient audience? We need an audience with human knowledge who is willing to give feedback about the meaningfulness of many sentence fragments. In practice, the texts accessible on the Internet represent a huge source of knowledge meaningful for humans. We use these texts to play the role of the knowledge base of our audience. We utilize the power of web search engines to validate the utterances. Whenever the search engine returns a certain number of hits, we conclude that the sentence fragment is meaningful for humans.

Contributions and outline: In Section II, we present several forward engineering guidelines for transforming natural language texts into object-oriented design and further into programs. Section III represents the core of this paper, describing a new method to extract fragments of domain knowledge from programs (knowledge quarks) by transforming program source code into fragments of natural language sentences. In Section IV, we present several case-studies on applying our program analysis approach to parts of the Java standard API and an open source Java program. We focus on several applications: (1) identifying the complex concepts contained in compound identifiers, (2) identifying unintuitively named classes, and (3) extracting complex relations between concepts implemented in the code. In Section V, we present a discussion of the limitations and variability points of our approach. Related approaches for analyzing code are presented in Section VI. We conclude the paper and present our plans for future work in Section VII.

II. From Natural Language to Programs

In this section, we briefly present guidelines of object-oriented modeling and design approaches for transforming parts of the requirements written in natural language texts into program abstractions during forward engineering.

In Section III, we present our approach to do the inverse way and transform the source code into natural language sentence fragments by starting from program abstractions.

Numerous software methods as well as design and development books describe the process for the transition from the knowledge about the business domain (as presented in textual requirements) to the design [1], [2], [12], [13].

A pioneering work for program design, starting from informal descriptions of a problem domain given in English, dates as early as the beginning of the 1980s:

> "Most striking is the parallel between the **noun phrases** in the original and the **data types and program objects** in the program. The bulk of this article is a discussion of that correspondence and of how to make the transformation between noun phrases and data types and objects." [13] (our emphasis)

Object-oriented programming is supposed to intuitively represent domain concepts in code. For example, in the time of the advent of object-orientation, its promoters advocated the naturalness and correspondence between domain concepts as expressed in natural language and the object-oriented code:

> "Through the design activity, the structure of software modules based on an object oriented model is interactively extracted from the informal English description. Each word such as **nouns** and **verbs** in the **natural-language sentences** is associated with a software concept, e.g. **class**, **attribute**, and **method**." [12] (our emphasis)

Another example is in the following:

> "Many objects are there just for the picking. They directly model objects of the physical reality to which the software applies. [...] using **software object types (classes)** to model **physical object types**, and the method's inter-object-type relations [...] to model the relations that exist between physical object types such as aggregation and specialization." [1] (our emphasis)

Considerable work has been done in forward engineering that guides the extraction of object models from natural language texts. However, the potential to help reverse engineering by performing the backwards way, i.e. extracting natural language texts from program code, is much less investigated (to the best of our knowledge not at all).

We present a method for extracting domain knowledge from code by recovering sentence fragments that could have been used as an input for creating the code.

III. From Programs to Natural Language

Approach Overview: In Figure 3, we present an overview of our approach and the basic components needed to obtain semantic information from programs. The main component of our toolkit is the semantic program interpreter. It takes as an input a series of program facts (e. g. "class Rectangle extends Shape", "class Rectangle implements Cloneable") and a series of strategies for transforming these facts into fragments of English sentences (e. g. "a ∇ is a $\diamond$ that"). Using these two inputs, the interpreter generates a series of fragments of natural language sentences (e. g. "a rectangle is a shape that", "a rectangle is a cloneable that") and uses them as inputs for a web search engine. Depending on the number of hits returned by the web search engine (i. e. the number of times that sentence fragment was actually written by someone) the semantic interpreter finds a new relevant fact about the program with a higher or lower confidence. It might even reject that program fragment as not implementing common knowledge known to humans. Once identified, a knowledge fragment is saved into a knowledge base.

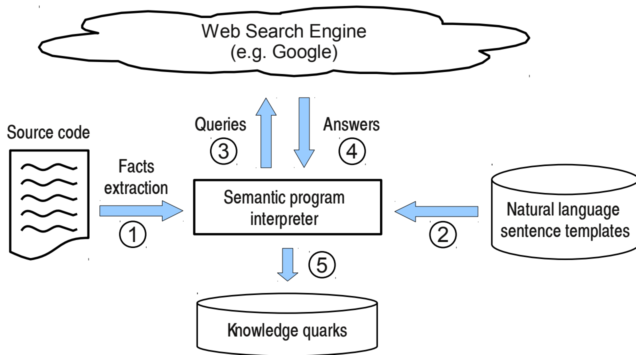

Figure 3. Overview of our approach

In Figure 4, we present a typical example of a program fragment containing different pieces of knowledge about the business domain (i. e. geometrical figures). In the following, we will use this example to illustrate how we abstract from program code, what sentence templates we choose, how we systematically transform the code into sentence fragments and how we perform queries.

```
class Rectangle implements Shape, Clonable {
    float currentWidth, currentHeight;
    void draw() { ...}
    Object clone() { ...}  ...
}
class GeometricalTransformationUtils {
    void rotate(Shape) { ...} ...
}
```

Figure 4. Typical program fragment example

In the following subsections we present each of the steps of our approach in more detail:

A. Extracting facts from Java programs

The biggest part of Java programs and the most important link to the domain knowledge implemented in the code is made of the identifiers defined by programmers. Relations between identifiers are defined by their corresponding program elements. Thereby, we abstract programs as sets of identifiers and relations between them.

Program We consider a program P to be a tuple

$$P = (I, R)$$

whereby I is the set of program identifiers and R is a set of typed program relations between the program elements corresponding to these identifiers. For the relations in R, we choose the following types: *hasSuperType, hasAttribute, hasMethod, hasParameter* with their obvious meanings.

Intuitively, we consider a program to be a graph whose nodes are identifiers and whose edges are the relations between the identifiers. In Figure 5, we present an example of the program graph that corresponds to the code fragment example from Figure 4. Each identifier can be formed of a single word or of a sequence of words that are not lexically normalized (e. g. we consider "rectangular", "rectangle", "rectangles" to be distinct words even if they have the same root).

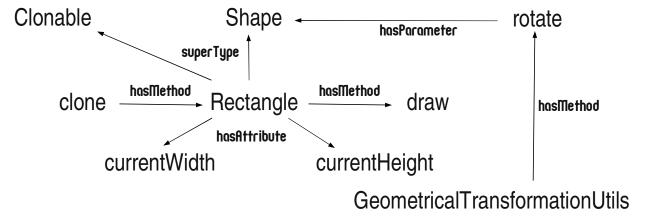

Figure 5. Example of the program graph

Identifier-to-words Given the set of identifiers I and the set of words W, we define the function

$$i_2w : I \rightarrow W^*$$

which transforms identifiers into a sequence of words.

Identifiers are split into parts according to the CamelCase notation (e. g. "$i_2w(ArrayList) = \langle "Array", "List" \rangle$), or based on other common word delimiters like for example underscores (e. g. $i_2w(current_element) = \langle "current", "element" \rangle$). In order to eliminate the irrelevant words, we do not consider pronouns, prepositions, conjunctions and words formed from only one letter.

B. Defining sentence templates

Sentence templates Let s_1, s_2, s_3 be strings. We call a string to be a well formed sentence pattern iff it has the following form:

$$"s_1 \quad \nabla \quad s_2 \quad \diamond \quad s_3"$$

Intuitively, a string is a valid pattern if it has two place holders (holes). The holes are denoted in the above formula through ∇ and ⋄ and they will be filled with words extracted from the program text. A sentence template is a basic fragment of a natural language sentence that would convey a certain information to its audience about the concepts denoted by words that replace ∇ and ⋄. For example, a simple sentence template is:

$$"a \ \nabla \ is \ a \ \diamond \ that"$$

Sentence fragment Let $w, w' \in W$ be two words, and p a sentence template. We call a string s to be a sentence fragment obtained from the template p iff:

$$s = p(\nabla \mapsto w, \diamond \mapsto w')$$

Intuitively, a sentence fragment is obtained from a sentence template by replacing the holes in the template (∇ and ⋄) with words. An example of the instantiation of the above sentence template into a sentence fragment would be: “a rectangle is a shape that".

In each of the following paragraphs, we present sentence templates that are targeted to cluster words that logically belong together and to extract logical relations between concepts.

1) Sentence templates for complex concepts: In natural language, the concepts that are frequently used have words attached to them. More complex concepts, that are more special or less frequently used, are formed of sequences of words. We call the latter *complex concepts*. They are formed by joining together two words w_1 and w_2. Examples of complex concepts are: “operating system”, “file system”, “input stream” or “round rectangle”. In order to test whether two words w_1, w_2 form a complex concept, we use the set of patterns P_1:

P_1 = {"the ∇ ⋄ is", "a ∇ ⋄ is” }

whereby ∇ takes the place of the first word in the complex word and ⋄ takes the place of the second word.

2) Sentence templates for taxonomical relations: An important relation between concepts is the superordinate/subordinate (is-a) relation. A simple sentence template that conveys the taxonomical relation between two concepts is:

P_2 = {"a ∇ is a ⋄ that”}

whereby ∇ is the placeholder for the sub-concept and ⋄ is the place holder for the super concept.

3) Sentence templates for “properties” or “parts”: Another important class of relations between concepts is the “property-of” or “part-of” relation (a.k.a. meronymy). In natural language, we use the same sentence to convey the information that a concept is a property or part of another concept:

P_3 = {"∇ of a ⋄", "∇ of the ⋄"}

whereby ∇ is the place holder for the property and ⋄ is the place holder for the concept that bears that property.

4) Sentence templates for patients of actions: Actions are described by using verbs in natural language. Actions are performed on objects – e. g. the action “draw” can be performed on “figures”. In natural language, a simple sentence pattern that conveys the information that a concept is an object upon which an action is performed is:

P_4 = {"to ∇ a ⋄"}

whereby ∇ is the placeholder for the action and ⋄ is the placeholder for the object upon which the action is performed.

C. Extracting facts

Voting function Let $Templ$ be a set of strings forming valid sentence templates. We define the function

$$vote : Templ \times Word \times Word \to \mathbb{N} \quad (1)$$

whereas $vote(t, w_1, w_2)$ returns the number of hits obtained by performing a web search with the sentence $t(\nabla \mapsto w_1, \diamond \mapsto w_2)$.

Fact extraction. Let $w_1, w_2 \in W$ be two words, $Patt$ a set of well-formed sentence patterns, and let the function $extract$ be defined as follows:

$$extract : \wp(Patt) \times W \times W \to \mathbb{N},$$

$$extract(patt, w_1, w_2) = max_{p \in patt}(vote(p, w_1, w_2))$$

Intuitively, the $extract$ function returns the maximum number of hits obtained from the search engine by successively forming sentences from a given set of patterns.

In the following paragraphs, we use the patterns defined in the previous sub-section to extract semantical information from the code.

1) Interpreting compound identifiers: Compound identifiers are identifiers made of several words. Some of these words logically belong together and form complex concepts and some are only implementation noise. Identifying the complex concepts that are contained in compound identifiers is a central issue in the analysis of identifiers.

For example, in the case of the identifier “GeometricalTransformationUtils” we have $i_2w(\text{“}GeometricalTransformationUtils'') = \{"geometrical", "transformation", "utils"\}$. From these

three words, the words “geometrical” and “transformation” belong together, whereas the word “utils” is only implementation detail. We propose the following heuristic for the identification of complex concepts:

Heuristic 1 (Complex concepts) Let $i \in I$ be an identifier whereas $i_2w(i) = \{w_1, ..., w_n\}, n > 1$. We call the word sequence $"w_l \; w_{l+1}", 1 \leq l < n$ the *complex concept* contained in the identifier i iff

$$\forall 1 \leq k < n, k \neq l :$$
$$extract(P_1, w_l, w_{l+1})) \geq extract(P_1, w_k, w_{k+1})) > 0$$

Intuitively, the complex concept is formed by the two words that return the highest number of hits when replaced in the set of sentence fragments P_1. In order to detect the complex concept contained in the compound identifier “GeometricalTransformationUtils” from the example above, we query Google by using the following set of sentences:

"the geometrical transformation is"
"a geometrical transformation is"
"the transformation utils is"
"a transformation utils is"

As we would expect, each of the first two sentences makes much more sense for the audience than the other two and therefore we identified the complex concept “geometrical transformation”. Once we identified a complex concept, we add its name to the set of known words:

$$W' = W \cup \{"w_l \; w_{l+1}"\}$$

In the subsequent analyses, we consider complex words by splitting an identifier into words. We extend the function i_2w to a new version i_2w^+:

i_2w(“GeometricalTransformationUtils”) =
{“Geometrical”, “Transformation”, “Utils”}
i_2w^+(“GeometricalTransformationUtils”) =
{“Geometrical Transformation”, “Utils”}

2) Interpreting the type hierarchy: In object-oriented languages, we model the is-a relations between domain concepts through the inheritance relations between their corresponding classes (or interfaces). In our program abstraction, a fragment of the type hierarchy is given by an edge in the program graph: “$identifier_1$ - hasSuperType - $identifier_2$” whereby $identifier_1$ is the name of the sub-class and $identifier_2$ is the name of the super-class. Many times, words contained in these two identifiers reference domain concepts that are in the sub-concept – super-concept relation.

Heuristic 2 (Taxonomy) Let $i, i' \in I$ be two identifiers, and $(i_1, hasSuperType, i_2) \in R$. The words $w \in i_2w^+(i)$, $w' \in i_2w^+(i')$ represent concepts that belong to a taxonomical relation iff:

$$extract(P_2, w, w') > 0$$

Intuitively, we identify a valid taxonomical relation between the (complex) words of two identifiers representing the sub-class and super-class iff, by performing the query using the set of patterns P_2, we obtain at least one hit for at least one pattern. For example, from the code from Figure 4, we build the following sentences

“a rectangle is a shape that”
“a rectangle is a clonable that”

As we would expect, only the first sentence fragment makes sense for the audience (and thereby the $extract$ function returns a number bigger than 0), while the other inheritance relation represents only implementation details and thereby does not contain domain related information.

3) Interpreting the attributes: In object-oriented languages, classes and attributes are frequently used to model the property or part relation between the concept implemented through the class and the one implemented as the attribute of that class. In our program abstraction, the relation between classes and their attributes is given by the following edge in the program graph: “$identifier_1$ - hasAttribute - $identifier_2$” Below, we present a heuristic for extracting the concepts and their properties.

Heuristic 3 (Parts and properties) Let $i, i' \in I$ be two identifiers, and $(i_1, hasAttribute, i_2) \in R$. The words $w \in i_2w^+(i)$, $w' \in i_2w^+(i')$ represent concepts among which there is a “property” relation iff:

$$extract(P_3, w, w') > 0$$

Intuitively, we identify a valid concept and its properties iff by performing the query using the set of patterns P_3, we obtain at least one hit for at least one pattern. For example, from the code from Figure 4, we extract the following sentence fragments:

“width of a rectangle” “width of the rectangle”
“height of a rectangle” “height of the rectangle”
“current of a rectangle” “current of the rectangle”

Here, only the first two sentence fragments are meaningful for the audience.

4) Interpreting the methods: In object-oriented languages, methods are typically used to represent actions in the system to be modeled. The objects on which these actions are performed (i. e. the patients of the actions) are implemented either as classes that contain the methods or as parameters of the methods. Given the edges “$identifier_1$ - hasMethod - $identifier_2$” or “ $identifier_1$ - hasParameter - $identifier_2$” in the program graph, we use the following heuristic to identify the actions and their objects:

Heuristic 4 (Objects of actions) Let $i, i' \in I$ be two identifiers, and $(i, hasMethod, i') \in R$ or $(i', hasParameter, i) \in R$. The words $w \in i_2w^+(i)$, $w' \in i_2w^+(i')$ represent concepts among which there is the "actsOn" relation iff:

$$extract(P_4, w', w) > 0$$

Intuitively, we identify a valid action and its object iff by starting from a class and a method (or a method and its parameter respectively) and performing the query using the set of patterns P_4 we obtain at least one hit. For example, from the code from Figure 4, we form the following sentence fragments:

"to draw a rectangle"
"to clone a rectangle"
"to rotate a shape"

As we would expect, we obtain hits only for the first and third sentences.

IV. Case-Studies

The central idea of our approach is that we can extract knowledge quarks by systematically transforming parts of the source code into fragments of natural language sentences. If these sentences are meaningful for humans, we identified a knowledge quark that is implemented in the code. The meaningfulness of these sentences can be tested by forming queries for a web search engine. This operationalization is based on two assumptions:

1) All texts indexed by Google are written by humans and make sense for humans. In other words: If Google returns a hit, the sentence fragment is meaningful.
2) Every meaningful sentence fragment that can be formed with our templates was written by someone in an electronic document that is indexed by Google.

The first assumption directly influences the precision of the approach for extracting facts from the code while the second assumption directly influences the recall. In our experiments, we only evaluated the precision.

Experimental setup: In order to perform the experiments, we chose to analyze different parts of the Java 1.5 standard API (i. e. the Collections API and the I/O API from the package `java.io`) and JHotDraw 5.4b1[1], a framework for drawing. The reason for this choice is that the Java Collections API implements data structures that represent one of the most well-understood and normed set of concepts in computer science. The I/O API implements a well-defined but larger set of concepts, and we consider JHotDraw to be a general purpose program. The version of the Collections API that we analyzed has 35 classes, the I/O API has 83 classes, and JHotDraw has 268 classes.

[1] http://www.jhotdraw.org/

We evaluate the feasibility of our approach for a set of use-cases that are closely related to the knowledge quarks that we extract from the code. Each of the following subsections is centered around a specific analysis use-case.

A. Interpreting compound identifiers

One of the most direct uses of our knowledge extraction framework is to interpret compound identifiers and to extract the complex concepts that are described by words that logically belong together.

Extracting complex concepts: In Figure 6, we present a list with the complex concepts that we automatically extracted from the classes, attributes and methods of the Java Collections framework. We extracted 65 complex concepts, out of which 7 represent false positives (these concepts are underlined in the figure and represent implementation decisions rather than concepts about data structures).

Complex concepts from the "collections" class names: linked list, random access, array list, priority queue, tree set, sub list, list iterator, tree map, hash map, sorted set, sequential list, abstract set, enum set, hash set, sorted map, abstract list, abstract collection, abstract map, abstract queue

Complex concepts from the "collections" attributes names: element type, empty list, reverse order, empty set, key set, key type, maximum capacity, load factor, zero length, serial version, entry set, element data, initial capacity, access order, empty map, minimum capacity, null key, element count, empty iterator, capacity increment, default capacity, empty enumerator

Complex concepts from the "collections" method names: null value, binary search, valid key, non null, hash code, tree set, key index, red level, sub list, hash map, sorted set, sequential list, preceding entry, search null, hash set, sorted map, eldest entry, all elements, value iterator, more elements, stale entries, ceil entry, key iterator, entry iterator

Figure 6. The complex concepts of the Java Collections API

The complex concepts have a high importance and raise the level of discourse at which a piece of software is analyzed – it is qualitatively different to regard the individual words "list", "array", "sub", "linked" or to say "array list", "sub list" or "linked list". Furthermore, complex words have a much lower ambiguity (polysemy) degree and can therefore be used directly for string-based concept location.

In the case of `java.io`, our tool automatically identified 188 complex concepts. An example of a complex concept is "file system" – the words "file" and "system" were found as a compound over nine million times. After their manual investigation, we identified that 43 (i. e. 23%) do not make sense. In the case of JHotDraw, our tool extracted 520 complex concepts; after their manual investigation, we identified that 75 (i. e. 14%) combinations of words do not make sense. We conclude that our method has a high precision for the extraction of complex concepts.

Assigning dominant complex concepts to compound class identifiers: In the following, we present a related use-case, namely the automatic assignment of dominant complex concepts to identifiers that contain more than two words. Due to space limitations, we present the exact assignment of complex concepts only for the classes from the Java

Collections API. In Figure 7, we present the interpretation of class names with compound identifiers containing at least three words. The figure contains four columns: the leftmost column contains the name of the class, the second column contains the sentence pattern that was used for the Google query, the third column contains the number of hits returned by Google, and the rightmost column contains the identified complex concept. We remark that in the first case, our tool identified that "random access" is a more important concept that "sub list" – we consider this situation to be flawed.

Program fragment	Sentence fragment	#Hits	Concept
class RandomAccessSubList	a random access that	202000	random access
class LinkedHashMap	a hash map that	162000	hash map
class WeakHashMap	a hash map that	162000	hash map
class IdentityHashMap	a hash map that	162000	hash map
class AbstractSequentialList	a sequential list that	47000	sequential list
class LinkedHashSet	a hash set that	21200	hash set

Figure 7. The complex concepts assigned to class names from the collections API

In case of the Java Collections API and JHotDraw, the manual inspection of the results showed that over 90% of the compound identifiers of class names were assigned correctly to complex concepts.

Identifying unintuitively named classes: In this use-case, we investigated the identification of classes whose names do not reflect common concepts. They are good candidates to be considered for choosing a more intuitive name. In the case of classes whose names are compound identifiers formed from two words, we would expect that these words were used together many times – this would show that the words logically belong together. In Figure 8, we present the class names, consisting of two words, to which no complex concept could be assigned.

"java.io" classes with unintuitive names: PipedReader, PipedWriter, SerializablePermission

JHotDraw classes with unintuitive names: AutoscrollHelper, AWTCursor, DesktopListener, DNDHelper, EastHandle, FigureEnumeration, FigureEnumerator, GridConstrainer, HandleEnumeration, HandleEnumerator, HTMLLayouter, PeripheralLocator, PointConstrainer, RelativeLocator, ScalingGraphics, SetWrapper, StandardLayouter, StorableInput, StorableOutput, UndoableAdapter, UndoableHandle, UndoableTool, VersionRequester, WestHandle

Figure 8. Classes with names consisting of two words that could not be assigned to complex concepts

By manually inspecting the classes PipedReader and PipedWriter from Java IO, we remarked that the programmer documentation (JavaDoc) talks about streams (Figure 9). However, the Input/OutputStream class hierarchies are complementary to those of Reader/Writer. The former works on byte arrays whereas the latter on character arrays. This fact is not reflected in the name of the classes.

In the case of the third class, the word "serializable" represents an implementation detail, and in this case its name is well chosen.

```
package java.io;
...
/**
 * Piped character-output streams. ...
 */
class PipedWriter extends Writer {
    …
```

```
package java.io;
...
/**
 * Piped character-input streams. ...
 */
class PipedReader extends Reader {
    …
```

Figure 9. Excerpt of the Javadoc for the classes PipedWriter and PipedReader

In the case of many classes from JHotDraw (Figure 8 - down) we remark that their names contain a word referring to a technical concept (e. g. "enumeration", "awt", "listener") and another word referencing a concept related to figures. There are however classes that refer to unusual concepts (e. g. "PointConstrainer", "UndoableTool"). These classes represent false positives, i. e. valid concepts that make sense for humans from the point of view of the JHotDraw domain knowledge. In Figure 10, we present an example of a class from JHotDraw whose name could not be assigned to a concept by using our method. By taking a closer look at the JavaDoc, we remark that a better name would be "ScalableGraphicsContext".

```
package CH.ifa.draw.contrib.zoom; ...
/**
 * A graphics context that can scale to an arbitrary factor.
 */
class ScalingGraphics extends … {    …
```

Figure 10. Example of a JHotDraw class with an unintuitive name

B. Extracting the intended taxonomical relations from the class hierarchy

In Figure 11, we present the extracted taxonomical relations from the class hierarchy of the Java Collections framework and JHotDraw. In both cases, the situations when we received only one hit from Google represent noise. We obtained the most important relations between the concepts of the Java Collections API. These are presented in Figure 6. However, some concepts from the taxonomy are missing, e. g. : "tree set", "array list". The cause for these omissions is the fact that Google does not index any document containing the text "an array list is a list that" or "a tree set is a set that". In case of Java IO, our method could not extract any taxonomical relation.

C. Extracting properties of concepts

We extracted the concepts and properties from the Collections API which contains 109 "class–attribute" relations. The tool automatically extracted 31 relations. After their manual inspection, we found that 19 (61%) relations are indeed meaningful (presented in Figure 12-left-up). In the same figure, below the line, we present several examples of relations that do not make sense – these results were

Taxonomy from Java Coll.	Taxonomy from JHotDraw	
Set – isA – Collection	Polygon – isA – Figure	Command – isA – Item
Queue – isA – Collection	Triangle – isA – Figure	Menu Item –isA– Command
List – isA – Collection	Event – isA – Event Object	Selection Tool – isA – Tool
Sorted Set – isA – Set	Ellipse – isA – Figure	Component – isA – Figure
Linked List – isA – List	Connection – isA – Figure	Mini Map –isA– Component
Hashtable – isA – Dictionary	Composite Figure –isA– Figure	Tracker – isA – Tool
Hash Set – isA – Set	Rectangle – isA – Figure	Null Handle – isA – Handle
Sorted Map – isA – Map	Command Button –isA– Button	Check Box – isA – Item
Vector – isA – List	View – isA – Component	Creation Tool – isA – Tool
Stack – isA – Vector	Triangle –isA– Geometric Figure	Action Tool – isA – Tool
Hash Map – isA – Map	Polygon –isA– Geometric Figure	
Abstract Map – isA – Map	Ellipse – isA – Geometric Figure	

Figure 11. Extracted taxonomies

obtained by finding the sentence fragments that are parts of longer sentences (e. g. there is a web page containing the fragment "size of the weak link" and thereby the tool mistakenly identified that size is a property of weak).

In case of the IO API, we analyzed 326 "class – attribute" relations. The tool automatically extracted 118 "hasProperty" relations. The manual inspection revealed that 58 (49%) of the relations are meaningful.

D. Extracting actions and their objects

In case of the Collections API, we investigated 1070 relations between classes and methods ("hasMethod" edges in the program graph) and between methods and their parameters ("hasParameter" edges). The tool automatically identified 107 "actsOn" relations. The manual inspection revealed that 70 (65%) of these relations are meaningful. In Figure 12-right, we present examples of good relations (upper part) and bad relations (lower part). We remark that in the case of good relations, even if they are meaningful, some of them do not reflect the intended meaning in the code (e. g. "Hashtable" is usually not an object of the action "put", but rather its doer. In other words, instead of "Put – actsOn – Hashtable" we should rather have "Hashtable – isDoer – Put").

In case of the IO API, we analyzed 1300 "hasMethod" and "hasParameter" edges of the program graph, and the tool extracted 188 "actsOn" relations. The manual investigation revealed that 90 (48%) make sense from the point of view of the IO knowledge.

In the case of the analysis of actions and their objects, we have the same source of noise as presented above, namely that the searched fragments represent pieces of already existing sentences. For example, our tool identified the relation "is – actsOn – collection", since the web pages found by Google contain the sentence fragment "to is a collection" which is meaningless per se. The manual investigation revealed that such pages refer to completely other concepts and that they contain text like "This How To is a collection of important steps". Another source of noise is the fact that, in case of many words, the instantiation of the sentence templates provide sentence fragments that do not have a self contained meaning. For example, we wrongly identified the relation "read – actsOn – random" since the sentence fragment "to read a random" occurs on the Internet as sub-sentence of other fragments like for example ""to read a random access file", or "to read a random bit".

Properties from Java Collections	Actions and objects from Java Collections
Array List – hasProp – Size	Add – actsOn – Collection
Collections – hasProp – Copy	Clear – actsOn – Collection
Hash Map – hasProp – Initial Capacity	Contains – actsOn – Collection
Hash Map – hasProp -- Load Factor	Empty – actsOn – Collection
Hash Map – hasProp -- Size	Remove – actsOn – Collection
Hashtable – hasProp – Entries	Retain – actsOn – Collection
Hashtable – hasProp – Keys	Empty – actsOn – Dictionary
Hashtable – hasProp – Load Factor	Put – actsOn – Dictionary
Hashtable – hasProp – Values	Remove – actsOn – Dictionary
Linked List – hasProp – Header	Add – actsOn – Enum Set
Linked List – hasProp – Size	Copy – actsOn – Enum Set
Map – hasProp – Array	Create – actsOn – Hash Map
Map – hasProp – Key	Put – actsOn – Hashtable
Map – hasProp – Key type	Remove – actsOn – Index
Map – hasProp – Size	Push – actsOn – Item
Map – hasProp – Universe	Remove – actsOn – Iterator
Map – hasProp – Vals	Map – actsOn – Key
Priority Queue – hasProp – Initial Capacity	Mask – actsOn – Key
Priority Queue – hasProp – Size	Remove – actsOn – Key
Vector – hasProp – Element Count	Add – actsOn – Linked List
Vector – hasProp – Element Data	Clear – actsOn – Linked List
Weak – hasProp – Size	Is – actsOn – Collection
Weak – hasProp – Threshold	Clone – actsOn -- Identity
Linked – hasProp – Header	Create – actsOn -- Identity

Figure 12. Extracted properties (left) and actions (right) from Java Collections

V. Discussion

1) On the fundamental limitations of our approach: The most important limitation is that programs contain knowledge besides the one that can be expressed through simple sentence templates. For example, none of our templates captures complex causality relations, behavioral dependencies between different domain concepts, or other complex relations. Such knowledge is neither captured by our program abstraction (i. e. identifiers and relations among them) nor is it expressible through the sentence templates that we used. Therefore, the use of our method cannot extract the whole knowledge that is implemented in the code but only a subset thereof.

2) On the limitation of using pre-defined templates: We assume that we have a large enough basis of common knowledge, available as natural language texts, and that the knowledge is expressed through our defined patterns. Even if the results of our case-studies suggest that many times this is the case, we also discovered considerably many cases when a certain knowledge quark, even if it is contained in the WWW, it is not available in the predefined sentence patterns. We also remarked that the results are highly sensitive with respect to the exact string we use for query. For example, the query "key set of a hashtable" returned no hit (at the date we performed the query, namely

19.02.2010) while the query "key set of the hashtable" returns several hits. Therefore, to obtain a high recall, we need to use a more extended set of sentence patterns with different slight variations.

Our set of heuristics is not comprehensive and should be extended with more templates for relations or even with new templates that capture other logical relations among implemented concepts. We foresee that this can be a fruitful research direction at the intersection between linguistics and reverse engineering.

Another limitation is that we often obtain ambiguous results (as we explained in detail in Section IV-C and IV-D). A possible solution is to define more restrictive patterns – however, in these cases we would lower the recall.

3) On the particularities of the Google search engine: Besides the limitations due to the knowledge kinds that we address and due to the pre-defined patterns (which generate false negatives), there are limitations due to the use of the Google search engine. For example, Google removes the punctuation marks when performing the queries. This fact generates unexpected results – e. g. the query "to read a child" can return a web page containing the following text: " ... I urged him to read. A child just came in ..." which is clearly not the intended meaning of the search query. This leads to false positives.

We assumed that the higher the number of hits returned by Google, the more frequent the sentence fragment occurs in the World Wide Web and thereby the more relevant the extracted fact. Even if this proves to be true in most cases, (e. g. "file name" or "linked list" return many hits) there are however exceptions since the number of hits returned by Google following a certain query is only a rough approximation of how many times a certain text is really written in the web-pages. However, the use of different threshold values and their influence on the precision and recall is a subject of further investigations.

4) On the ambiguous nature of the natural language: Natural language words are ambiguous since they exhibit synonymy and polysemy. By searching for sentence parts, we restrict the ambiguity to a great degree. However, by using only simple sentence fragments we cannot eliminate the semantic ambiguity completely – e. g. when analyzing the program fragment "class Button extends Component", we extracted the sentence "a button is a component", and, as expected, Google returns many hits. Among these hits, there are also irrelevant documents that contain (by chance) the same text that refers to completely other concepts – e. g. "a button is a component of the electrical plan of the pumping station". These situations generate false positives since they obviously do not correspond to the intended meaning of the concepts implemented in the code.

5) On using other textual resources: In our experiments, we made use of Google searches on the whole World Wide Web. For this purpose we obtained a grant from Google that allows us to programmatically perform queries. However, even without access to Google, there are other Internet resources that could be used as huge textual bases. An example would be to use the freely available electronic encyclopedia Wikipedia[2] as a text basis on which searches can be performed (the archive with Wikipedia can be downloaded and installed locally on a computer with appropriate resources). Furthermore, by performing scoped searches (i. e. by taking into account only certain web pages like those that have a proven quality), we can increase the accuracy (with the danger to lower the recall).

6) On using manual inspections for evaluation: We used our subjective judgement for evaluating the meaningfulness of the extracted facts. Therefore the result of the manual inspections might be biased.

VI. Related Work

To the best of our knowledge, there is currently no other reverse engineering approach that extracts facts from the code by transforming the code into natural language sentences and testing their meaningfulness. There are however different other related areas as presented below:

Concept location and concept assignment: Concept location [10] and concept assignment [11] are two central problems for program understanding. According to Biggerstaff et al., concept assignment can be split into "programming oriented concepts" and "human oriented concepts", the latter being more challenging to recognize in programs. The tasks required for the assignment of human oriented concepts are *"1. identify which entities and relations out of the overwhelming numbers in a large program are really important; 2. assign them to [...] domain concepts and relations"*[11]. Our work is nothing else than an operationalization of these tasks: we define the sentence templates to reflect the most likely relations between the implemented concepts and use the web search engine queries to identify which of these concepts are logically related and which relations make sense. Generally, the method presented in this paper extends the string matching methods for locating concepts with a more complex interpretation of the code.

Interpreting compound identifiers at semantical level: The linguistic content of compound identifiers was analyzed by [14] in the case of class names and by [15] in the case of function names. In the guidelines for good naming of classes developed by [14], the noun-noun compounds play a central role. Our experiments confirmed this fact, as many of the complex concepts are formed of two nouns. However, we discovered that a considerable number of complex concepts have the form adjective – noun (e. g. "sub list", "sequential list").

[15] analyze the structure of function identifiers and developed a sophisticated grammar representing the composition of function identifiers.

[2] www.wikipedia.org

We advance with respect to this work by interpreting the compound identifiers from a semantic point of view. Our approach can also be used to automatically test to what extent the compound class and function names respect identifier forming guidelines.

Using domain knowledge for program analysis: This paper is in-line with our previous line of research about interpreting code from the point of view of domain knowledge that it implements and using domain knowledge for program analysis [16]. Instead of mapping programs on ontologies, as we did in our previous work [17], we take a new approach and map program fragments on web resources that they are likely to implement. In the current work, the semantic domain is represented by the natural language texts available on the Internet and the interpretation function is the link between the program fragments and the web pages that contain the same domain knowledge. We consider these approaches to be complementary. Furthermore, the use of web search engines for extracting domain knowledge can help in the construction of an initial version of the domain ontology for a program, which can subsequently be used in further analyses.

VII. Conclusions and Future Work

Our experiments show that it is feasible to transform code into fragments of natural language sentences and use web search engines to validate their meaningfulness. Our approach can therefore be used to extract logical information from programs.

Having a method to interpret the identifiers at the semantic level is not our ultimate goal. We rather regard our method as an enabler for enhancing other program understanding activities. We presented our experience with performing logical analyses, including: interpretation of compound identifiers, identification of unintuitive naming, and recovery of logical relations between concepts implemented in the code.

A direction for future work is to improve the precision and recall of the extraction of concepts by defining other sentence templates (e. g. based on other program relations) or refining the current ones. Another direction is to further investigate how the approach can help typical program comprehension tasks and to define logical analyses at the code level based on the recovered knowledge.

VIII. Acknowledgments

This research draws on data provided by the University Research Program for Google Search, a service provided by Google to promote a greater common understanding of the web.

References

[1] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PTR, March 2000.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[3] B. Venners, “Think of Objects as Machines,” *Artima Developer, http://www.artima.com/*, 2003. [Online]. Available: http://www.artima.com/objectdesign/machine.html

[4] M. Henning, “Api design matters,” *ACM Queue*, vol. 5, no. 4, pp. 24–36, May/June 2007.

[5] J. Bloch, “How to design a good api and why it matters,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA’06)*. New York, NY, USA: ACM Press, 2006, pp. 506–507.

[6] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] S. Letovsky and E. Soloway, “Delocalized plans and program comprehension,” *IEEE Software*, vol. 3, no. 3, pp. 41–49, 1986.

[8] S. Rugaber, K. Stirewalt, and L. M. Wills, “The interleaving problem in program understanding,” in *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE’95)*, 1995.

[9] D. Ratiu and F. Deissenboeck, “From reality to programs and (not quite) back again,” in *Proceedings of the 15th International Conference on Program Comprehension (ICPC ’07)*, 2007.

[10] V. Rajlich and N. Wilde, “The role of concepts in program comprehension,” in *Proceedings of the 10 th International Workshop on Program Comprehension (IWPC’02)*, 2002.

[11] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” in *Proceedings of the 15th International Conference on Software Engineering (ICSE ’93)*. IEEE CS, 1993, pp. 482–498.

[12] M. Saeki, H. Horai, and H. Enomoto, “Software development process from natural language specification,” in *Proceedings of the 11th International Conference on Software Engineering (ICSE ’89)*. New York, NY, USA: ACM, 1989, pp. 64–73.

[13] R. J. Abbott, “Program design by informal english descriptions,” *Communications of the ACM*, vol. 26, no. 11, pp. 882–894, 1983.

[14] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, September 2006.

[15] B. Caprile and P. Tonella, “Nomen est omen: Analyzing the language of function identifiers,” in *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE ’99)*, 1999, pp. 112–122.

[16] D. Ratiu, “Intentional meaning of programs,” Ph.D. dissertation, Technische Universität München, submitted Feb 2009.

[17] D. Ratiu and F. Deissenboeck, “Programs are knowledge bases,” in *Proceedings of the 14th International Conference on Program Comprehension (ICPC ’06)*, 2006.