Automatic Extraction of a WordNet-Like Identifier Network from Software

Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, Clémentine Nebut, Violaine Prince, Michel Dao

Automatic Extraction of a WordNet-like Identifier Network from Software

J.-R. Falleri
RMoD Team
INRIA Lille Nord Europe
jean-remy.falleri@inria.fr

M. Huchard, M. Lafourcade, C. Nebut, V. Prince
LIRMM, CNRS and Université Montpellier 2
Montpellier, France
{huchard,lafourcade,nebut,prince}@lirmm.fr

M. Dao
Orange Labs
Issy-Les-Moulineaux, France
michel.dao@orange-ftgroup.com

***Abstract*—Softwares are designed to be used a significant amount of time, therefore maintenance represents an important part of their life cycle. It has been estimated that a lot of the time allocated to software maintenance is spent on the program comprehension. Many approaches using the program structure or external documentation have been created to ease the program comprehension. However, another important source of information is still not widely used for this purpose: the *identifiers*. In this article, we propose an approach, based on Natural Language Processing techniques, that automatically extracts and organizes concepts from software identifiers in a WordNet-like structure: lexical views. Those lexical views give useful insight on an overall software architecture and can be used to improve results of many software engineering tasks. The proposal is validated on a corpus of 24 open source softwares.**

I. Introduction

Softwares are designed to be used a significant amount of time, therefore maintenance represents an important part of their life cycle. In [1], it is estimated that up to 60% of the time allocated for software maintenance is spent on the comprehension of the program. A lot of work has been done to recover important insight about a program by the analysis of its structure or its external documentation. Unfortunately, a fundamental other source of information present in softwares is still not widely used: *the identifiers*. The identifiers are short names given by the developers to the different elements (functions, classes, attributes, types, variables) defined in a software. They constitute about 33% of all tokens [2] in the source code of softwares like Eclipse, Sun JDK or Tomcat. They also have a strong impact on the program comprehension.

In most of the existing work using software identifiers, they are considered as *bags of words*. On the contrary, we consider them as short sentences, containing words of distinct importance. For instance, let us consider these three identifiers: *OrderedSet*, *HashSet* and *setName*. After tokenization, these identifiers become: *ordered set*, *hash set* and *set name*. With no further analysis, these three identifiers seem highly related, because they all share the word *set*. Unfortunately, only the two first identifiers are related. Moreover, the *bag of words* paradigm is unable to extract hierarchical relations between the identifiers. Let us consider the two following identifiers: *List* and *LinkedList*. It is clear that the meaning of *List* is more general than the meaning of *LinkedList*.

Disposing of a lexical network like WordNet [3] is very useful to navigate through a set of words. Moreover, it can be used as the basis to compute several *semantic* similarity measures [4]. In such a lexical network, words are grouped into sets of words having the same meaning (*synsets*) and connected by several distinct relations (like *is more general than* or *is a part of*). The main problem is that obtaining such a lexical network usually requires a lot of manual work done by many domain experts. For instance, the original WordNet, that displays the relations between the general English words, has required several years of work. Having available such a lexical network for a particular software could ease many forward and reverse engineering tasks such as program comprehension, program visualization, naming assistance, architecture recovery or aspect mining. Unfortunately, creating it by hand would be very time consuming. Moreover, using the original WordNet would probably lead to poor results for several reasons. First, working in the context of a software system strongly restricts the semantic field, and some software terms, or term meanings cannot be found in WordNet: for example, is not included in WordNet (3.0 online search), and the computer science meaning of is not mentioned. Second, WordNet mostly includes atomic words, while identifiers are small sentences. Therefore, it is not clear how one can attach a given identifier in WordNet.

In this paper, we propose a novel approach that automatically classifies a set of identifiers in a WordNet-like structure. We call this structure a *lexical view*. It highlights the hierarchical relations between the identifiers. Moreover, it includes several implicit concepts that have been automatically extracted from the initial set of identifiers. This approach, that adapts and uses techniques from the Natural Language Processing (NLP) field, performs the following successive steps:

1) cut up identifiers in order to find the primitive words they are composed of,
2) classify the previously extracted primitive words into lexical categories (noun, verb, adjective, ...),
3) apply rules specific to the English language to determine which words are dominant and impose the meaning of the identifiers,

4) extract implicit important words,
5) organize the initial identifiers together with the freshly extracted words in a WordNet-like lexical view.

Our approach can be applied at any level of granularity: one can compute a lexical view of *class identifiers*, of *attribute identifiers*, etc. Since the only thing required by our approach is a set of identifiers, we can extract lexical views from softwares defined using a wide range of languages like object-oriented or functional languages.

In this paper, we first introduce our lexical views, and the underlying mathematical model we defined (section II). Then we detail our approach to extract the lexical views (section III). We show the validity and the interest of the approach by applying it on identifiers taken from 24 real-world Java programs (section IV). Section V discusses the related work and section VI describes the future work and concludes.

II. Lexical view foundations

Before precisely describing the foundations of our lexical views, we first give some insight about WordNet that has largely inspired our approach. WordNet [3] is a manually built lexical dictionary that displays the relations between the words. There are several differences between WordNet and a traditional dictionary. First, the words are grouped into synonym sets (called *synset*) instead of being displayed in alphabetic order. This has been done because the designers of WordNet make the difference between the syntax of a word (its *form*) and its semantics (its *meaning*). A synset is a set of words that have a different syntax but the same meaning. Second, the different words displayed in WordNet are split according to their grammatical categories (verb, noun, adjective). It means that a synset contains only words of the same grammatical category. Finally, these synsets are involved in several binary relations such as *is more general than* or *is a part of*. The relations allowed to connect synsets are constrained by the grammatical category of these synsets. Moreover, these relations can connect only synsets having the same grammatical category.

In this section, we define a precise model of such a lexical dictionary, called a *lexical view*. Our objective being to build a WordNet of identifiers coming from a software, we have several differences with the original WordNet. First, identifiers are short sentences and not atomic words. Therefore, we will base our model upon the notion of sentence. Second, since a sentence cannot easily be classified in a grammatical category, we do not separate them this way. Finally, since we focused in this paper on how to *automatically* extract the synsets and the relations of our lexical view, we limit the number of possible relations between the synsets to two: *hyperonymy* and *hyponymy*. These relations can be defined respectively as *is more general than* and *is more specific than*. We choose these relations for two reasons. First they are the only relations that can be safely derived from the identifiers. Second because these relations are the most important: in [4] we can see that most of the semantic similarity measures based on WordNet use only these relations.

In this section, we denote by $\mathcal{C}$ a strict partial order of concepts ordered by $<_c$, and $\mathcal{S}$ a set of sentences. The concepts represent the different possible meanings of the sentences. The order relation between the concepts represents the *is more specific than* relation. We assume that every sentence $s \in \mathcal{S}$ refers to a concept $c \in \mathcal{C}$. We introduce the sem function that maps every sentence to its referred concept, $sem : \mathcal{S} \rightarrow \mathcal{C}$. In the natural language, sem is a multivalued function: it is usual that a sentence $s \in \mathcal{S}$ can be mapped on two different concepts, given the context. For instance *condition* is mapped to the *disease* concept when one talks about medicine, but is mapped to another concept in the general case. Since we are working with identifiers coming from softwares, we constrain the sem to be a function, therefore a given sentence $s \in \mathcal{S}$ is mapped to only one concept $c \in \mathcal{C}$. We did that because usually words in software are used carefully. In our opinion it is very unlikely that a term appearing in a software with the same lexical catecory can have two different meanings (like *set* as a noun or *set* as a verb).

Definition 1: **Synonymy (syn).** Let $(s_1, s_2) \in \mathcal{S}^2$. We have $syn(s_1, s_2)$ iff $sem(s_1) = sem(s_2)$. syn indicates that two sentences have the same meaning.

Property 1: syn is trivially reflexive, symmetric and transitive. Therefore, it is an equivalence relation over $\mathcal{S}$.

Since syn is an equivalence relation over $\mathcal{S}$, we can derive the following property:

Property 2: syn defines several equivalence classes over $\mathcal{S}$.

We denote by $[s]$ the equivalence class of a sentence $s \in \mathcal{S}$. As previously seen, such an equivalence class is often referred to as *synset* in the WordNet litterature. We now define the *hyponymy* and *hyperonymy* relationships as:

Definition 2: **Hyponymy (hypo) and hyperonymy (hyper).** Let $(s_1, s_2) \in \mathcal{S}^2$. We have $hypo(s_1, s_2)$ (resp. $hyper(s_1, s_2)$) iff $sem(s_1) <_c sem(s_2)$ (resp. $sem(s_2) <_c sem(s_1)$).

Property 3: $hypo$ is the inverse relation of $hyper$.

Property 4: $hypo$ and $hyper$ are (by construction) irreflexive, asymmetric and transitive. They define a strict partial order over $\mathcal{S}$.

We have seen that syn defines equivalence classes in $\mathcal{S}$, we can therefore derive the following property:

Property 5: Let $(s_1, s_2) \in \mathcal{S}^2$. $hypo(s_1, s_2) \rightarrow \forall s_i \in [s_1], \forall s_j \in [s_2], hypo(s_i, s_j)$. The same property also applies for $hyper$.

Lexical views are a mean to display a set of sentences $\mathcal{S}$ together with their relations syn, $hypo$ and $hyper$. A lexical view lv is a labeled and directed graph $lv = (N, A)$ with N a set of nodes and A a set of directed edges. To

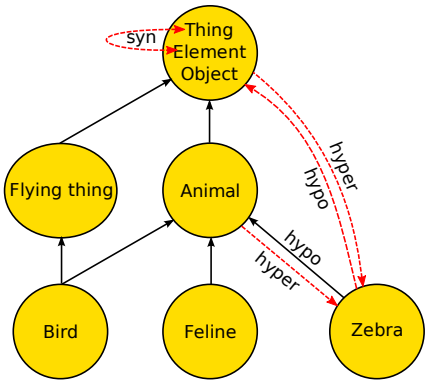


Figure 1. A sample lexical view

build a lexical view, a triple $(\mathcal{S}, syn, hypo)$ is required. Since $hypo$ and $hyper$ are inverse relations, it can also be done with a triple $(\mathcal{S}, syn, hyper)$. The nodes $n \in N$ of the lexical view represent the equivalence classes (synsets) defined by syn in $\mathcal{S}$. Edges $a \in A$ between the nodes of the lexical view represent $hypo$ relations between the sentences. Since the $hypo$ relation is transitive, the edges of the lexical view are created using the transitive reduction [5] of the $hypo$ relation. A lexical view is therefore very similar to the Hasse diagram of a partially ordered set, with the nodes representing equivalence classes instead of elements. Figure 1 shows a sample lexical view. Some of the relations that can be inferred from this view are represented by the dotted edges.

In this mathematical model, we defined the syn relation and how it can be displayed in the lexical view. However, we will not use it in our automatic extraction process. Nevertheless, we did define this relation because we think it is very useful in the case of a semi-automatic approach where a user-defined list of synonyms can be manually added to the input of the extraction process.

III. Automatic Extraction of Lexical Views

In this section we describe our approach that automatically computes a lexical view from a set of identifiers. In this lexical view, there is a node for each concept. Since we concentrated on the automation of the approach, our process currently does not let the user manually give synonymy information, thus, more precisely, there is a node for each identifier. Additionally, there are nodes for each implicit concept introduced by the identifiers. For instance, with two identifiers *HashSet* and *OrderedSet*, we build a view with three nodes, two nodes for the initial identifiers, and one additional node for the *Set* concept, implicitly introduced. This approach adapts and uses common techniques coming from the Natural Language Processing (NLP) field. It is composed of five steps:

1) **tokenization:** the identifiers are split into a list of terms,
2) **part-of-speech tagging:** part-of-speech (POS) types (verb, noun, ...) are assigned to each term of the previously computed lists,
3) **dependency sorting:** every tagged list of terms is sorted by the dominance order of the terms,
4) **lexical expansion:** implicit concepts are extracted from the previously sorted lists of tagged terms,
5) **lexical view computation:** a lexical view is computed, using concepts directly extracted from the identifiers and concepts discovered during the lexical expansion step.

A. Identifier tokenization

This step aims at extracting the terms that have been aggregated in the identifiers. For instance, *getNextWarning* should be decomposed into *get*, *next* and *warning*. These terms are often highlighted with a case change or special characters like underscore or hyphen, usually following team conventions and programming language rules.

We defined identifier tokenization as follows. Let I be the set of identifiers provided as the input of our process. T is the set of terms appearing in the identifiers. We define tokenization (tok) as a function taking as input an identifier and producing as output a list of terms of an arbitrary length. More formally,

$$tok : I \rightarrow \bigcup_{i=1}^{\infty} T^i \quad (1)$$

To create the list of terms $\tau = t_1, \ldots, t_k$ from an identifier i, we designed several simple algorithms exploiting heuristics based on how the developers build their identifiers. The clues we take into account are:

- Case changes (as in *getNextWarning*)
- Sequences of numeric characters (as in *block129*)
- Sequences of non alpha-numeric characters (as in *next_warning*)

Our algorithm reads sequentially the characters of i and performs a cut when one of the previous situations is triggered. The cut segment is then added at the end of τ and the algorithm resumes reading the identifier. This procedure stops when the end of i is encountered.

At the end of this step, the tok function has been applied to every identifier of I yielding a set L of term lists. L will be now the input of the next step, called part-of-speech tagging.

B. Part-of-speech tagging

In this second step, we apply a tag operation on every list of terms $\tau \in L$. This tag operation aims at assigning a part-of-speech (POS) to every term $t \in \tau$. The parts-of-speech are several lexical categories under which the words of a sentence can be classified. Two words with the same part-of-speech share common properties in the sentence. For

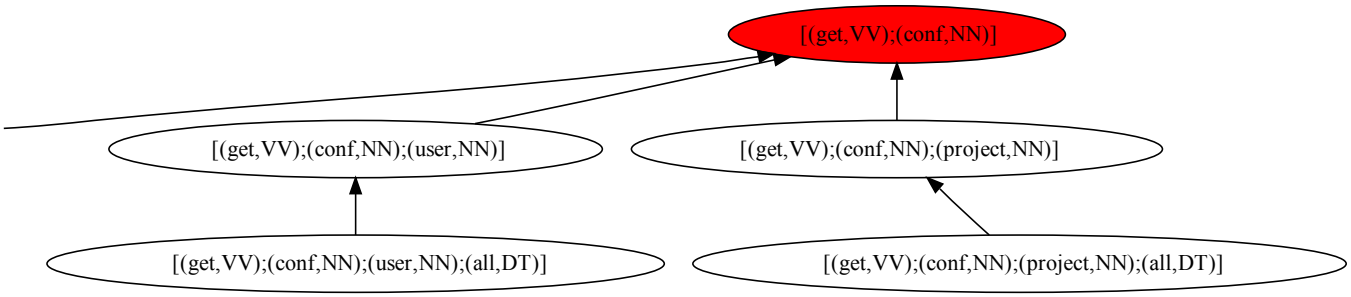


Figure 2. Excerpt of the operation lexical view of *Salome-TMF*

instance *noun*, *verb* or *adjective* are parts-of-speech. We denote by P the set of parts-of-speech. More formally,

$$tag : \bigcup_{i=1}^{\infty} T^i \rightarrow \bigcup_{i=1}^{\infty} (T \times P)^i \tag{2}$$

Computing the part-of-speech of a word within a sentence is a very complex operation that has been thoroughly investigated in the NLP field. Here the different lists of terms τ are our sentences. We use the tool *Tree Tagger* [6] to perform the part-of-speech tagging operation on τ. Tree tagger produces as output a list pt of pairs derived from τ. Let $\tau = t_1, \ldots, t_k$ be a list of terms. pt is a list $(t_1, p_1), \ldots, (t_k, p_k)$, with $p_i \in P$ being the part-of-speech corresponding to t_i.

We used the English version of *Tree Tagger*, therefore our parts-of-speech P are adapted to the English language. In this case, four categories of nouns are defined, whether they are common (NN) or proper (NP), singular or plural (S added to the category name). *Tree Tagger* also uses 18 verb categories, including VV for infinitive and VVN for past participle. Other categories of interest for us are adjective (JJ), comparative (JJR), preposition to (TO), preposition around, toward (IN), symbol (SYM) and number (CD). With these notations *get*,*next*,*warning* would be transformed into *(get,VV)*,*(next,JJ)*,*(warning,NN)*.

We apply the tag operation to every list of terms $t \in L$. This leads to the creation of a set PT of lists of pairs pt previously described. In the next step, we apply the dependency sorting to every element $pt \in PT$.

C. Dependency sorting

This operation is inspired by the dependency analysis or parsing [7], [8], a well-known technique in the NLP field. Roughly, the traditional dependency parsing operation takes as input a part-of-speech-tagged sentence. It connects the word by links representing the relation *dominate* between the words. The links are then used to build a dependency tree. Let us imagine the following phrase: *I ate an orange and a green apple*. The verb *ate* applies on (dominates) *orange* and *green apple*, but the adjective *green* applies only on (is dominated by) *apple*.

Sentences derived from identifiers are much simpler than real sentences. They describe a precise and atomic concept. For this reason, there is no need for a tree structure to represent them. Instead, an identifier can be represented by a list of terms ordered by the order of domination among the terms: in such a list, a given term dominates all the terms located after him. For instance *get*,*next*,*warning* could be reorganized into *get*,*warning*,*next* because this identifier represents a *get* action applying on the *warning* which is *next* to the current one. We name this reorganization operation *dependency sorting*, or $dsort$. More formally,

$$dsort : \bigcup_{i=1}^{\infty} (T \times P)^i \rightarrow \bigcup_{i=1}^{\infty} (T \times P)^i \tag{3}$$

The dependency sorting is driven by an ordered set of rules, specifically designed for lists of terms with parts-of-speech in English. It takes as input an element $pt \in PT$ and produces as output an element pt'. The pt and pt' elements are given to a procedure which searches the first rule that applies. This rule is then applied, the corresponding action is performed, and the procedure is recursively called with the modified pt and pt'. We apply this operation on every element $pt \in PT$ to build a PT' set. More precisely, the set of rules that creates an element pt from an element pt' is the following (where $|pt|$ is the size of pt, i.e. its number of terms):

1) $|pt| = 0 \Rightarrow$ *stop*.
2) $|pt| = 1 \Rightarrow$ insert the element of pt at the end of pt', and remove it from pt.
3) $|pt| = 2$, the first element is a noun, while the second is not, $\Rightarrow$ the first element is added at the end of pt' and removed from pt.
4) the first element of pt is a verb $\Rightarrow$ it is added at the end of pt' and removed from pt.
5) the first element of pt is a preposition $\Rightarrow$ it is added at the end of pt' and removed from pt.
6) pt is the sequence (s = elements that are not prepositions, p = preposition, r = the rest) $\Rightarrow$ apply rules to s and add the result at the end of pt', then add p,

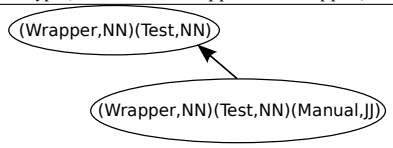
Step	Identifier 1	Identifier 2
	TestWrapper	*ManualTestWrapper*
Tokenization	*Test,Wrapper*	*Manual,Test,Wrapper*
POS tagging	*(Test,NN),(Wrapper,NN)*	*(Manual,JJ),(Test,NN),(Wrapper,NN)*
Dependency sorting	*(Wrapper,NN),(Test,NN)*	*(Wrapper,NN),(Test,NN),(Manual,JJ)*
Lexical expansion	$\emptyset$	
Lexical relations	*hypo(ManualTestWrapper,TestWrapper)*	
Lexical view		

Table I
ANALYSING TWO *wrapper* IDENTIFIERS (FROM *Salome-TMF*)

finally apply rules to r and add the result at the end of pt'.

7) the last element of pt is a number $\Rightarrow$ it is moved at the beginning of pt.
8) **(default rule)** the last element of pt is added at the end of pt' and removed from pt.

This set of rules might seem simplistic, but it captures most of the constructions that we found in the identifiers of a lot of softwares. Let us apply these rules to the identifier *JavaBlock12*, that has been tokenized and POS tagged to *(Java,NN),(Block,NN),(12,CD)*:

pt = *(Java,NN),(Block,NN)(12,CD)*; $pt' = \emptyset$

1. Rule 7 (last element is a number), move *(12,CD)* at the beginning of pt.
   pt = *(12,CD),(Java,NN),(Block,NN)*; $pt' = \emptyset$
2. Rule 8 (default), transfer *(Block,NN)* at the end of pt'.
   pt = *(12,CD),(Java,NN)*; pt' = *(Block,NN)*
3. Rule 8 (default), transfer *(Java,NN)* at the end of pt'.
   pt = *(12,CD)*; pt' = *(Block,NN),(Java,NN)*
4. Rule 2 ($|pt| = 1$), transfer *(12,CD)* at the end of pt'.
   $pt = \emptyset$, pt' = *(Block,NN),(Java,NN),(12,CD)*

Here is another example with the identifier *getChangeListener*, that has been tokenized and POS tagged to *(get,VV),(Change,NN),(Listener,NN)*:

pt = *(get,NN),(Change,NN),(Listener,NN)*; $pt' = \emptyset$

1. Rule 4 (first element is a verb), transfer *(get,NN)* to the end of pt'
   pt = *(Change,NN),(Listener,NN)*; pt' = *(get,NN)*
2. Rule 8 (default), transfer *(Listener,NN)* to the end of pt'.
   pt = *(Change,NN)*; pt' = *(get,NN),(Listener,NN)*
4. Rule 2 ($|pt| = 1$), transfer *(Change,NN)* to the end of pt'
   $pt = \emptyset$, pt' = *(get,NN),(Listener,NN),(Change,NN)*

D. Lexical expansion

During this step, we extract implicit concepts from the set of identifiers. These concepts, while present in the software, are never directly addressed by an existing identifier (like the *Set* concept that would have been induced by two *OrderedSet* and *HashSet* identifiers). During this step, a set PT'' is derived from PT', such as $PT' \subseteq PT''$. To create this PT'' set we use a lcp operation, that returns the longest common prefix between two elements of PT'. More formally, we have:

$$lcp : \bigcup_{i=1}^{\infty} (T \times P)^i \rightarrow \bigcup_{i=1}^{\infty} (T \times P)^i \quad (4)$$

Let

$$pt_1 = (t_0^1, p_0^1), \ldots, (t_k^1 p_k^1) \quad (5)$$
$$pt_2 = (t_0^2, p_0^2), \ldots, (t_m^2 p_m^2) \quad (6)$$
$$(7)$$

We have

$$lcp(pt_1, pt_2) = (t_0^1, p_0^1), \ldots, (t_l^1, p_l^1), l < min(k, m) \quad (8)$$

Such as

$$\forall i \in [0, l], (t_i^1 = t_i^2) \wedge (p_i^1 = p_i^2) \quad (9)$$

And

$$\text{if } l < min(k, m), \neg(t_{l+1}^1 = t_{l+1}^2) \vee \neg(p_{l+1}^1 = p_{l+1}^2) \quad (10)$$

We start with a PT'' set equal to PT', and we apply the lcp operation to every possible combination of two elements from PT''. Whenever $|lcp(pt_1, pt_2)| > 0$, we add $lcp(pt_1, pt_2)$ in PT'' if it is not already present. For instance, let us imagine the two following identifiers: *LinkedList* and *ArrayList*. After tokenization, POS tagging and dependency sorting, we have pt_1 =*(List,NN),(Linked,VVD)* and pt_2 = *(List,NN),(Array,NN)* in the PT'' set. We compute $lcp(pt_1, pt_2)$ = *(List,NN)*. Since there is no such element in PT'', *(List,NN)* is added to PT''.

E. Lexical view computation

In this step, we first build a graph $G = (N, A)$, with N a set of nodes and A a set of directed edges. This graph is computed from the elements in PT''. In our graph, there is a node for each element $pt \in PT''$ (therefore $|N| = |PT''|$).

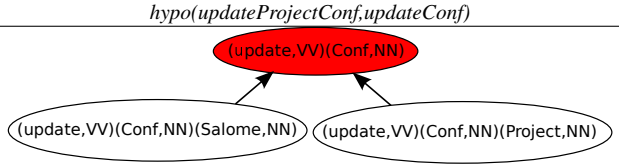
Step	Identifier 1	Identifier 2
	updateSalomeConf	*updateProjectConf*
Tokenization	*update, Salome, Conf*	*update, Project, Conf*
POS tagging	*(update,VV),(Salome,NN),(Conf,NN)*	*(update,VV),(Project,NN),(conf,NN)*
Dependency sorting	*(update,VV),(conf,NN),(salome,NN)*	*(update,VV),(conf,NN),(project,NN)*
Lexical expansion	*(update,VV),(conf,NN)*	
Lexical relations	*hypo(updateSalomeConf,updateConf)* *hypo(updateProjectConf,updateConf)*	
Lexical view		

Table II
ANALYSING TWO *update conf* IDENTIFIERS (FROM *Salome-TMF*)

We define $n : PT'' \rightarrow N$ the bijection that maps an element pt to its corresponding node $n \in N$. The edges in this graph represent the $hypo$ relations between the elements of PT''. We consider that an element pt_1 is an hyponym of an element pt_2 if pt_2 is a prefix of pt_1. For instance, it is reasonable to say that *(List,NN),(Linked,VVD)* is an hyponym of *(List,NN)*.

We first create nodes corresponding to every element of PT''. Then we apply the previous procedure lcp to every combination of two elements from PT''. Whenever $|lcp(pt_1, pt_2)| > 0 \wedge |pt_1| = |lcp(pt_1, pt_2)|$, we create an edge between $n(pt_1)$ and $n(pt_2)$.

Finally, we compute the transitive reduction of G [5]. This transitive reduction, denoted by lv, is our lexical view. Figure 2 shows the extract of a lexical view computed on identifiers coming from the *Salome-TMF* software. In this figure, nodes corresponding to the existing identifiers are represented in white. The concepts extracted during the *lexical expansion* are represented with a non-white background.

F. Examples

In this section, we show several small lexical view computation examples. We took these examples from real world software identifiers. Table I shows the lexical view computed from only two class identifiers coming from the *Salome-TMF* program: *TestWrapper* and *ManualTestWrapper*. In this table, results of the successive steps are given in the different rows. To assess if the produced lexical view was relevant, we looked in the code of *Salome-TMF*. There, we saw that the class *ManualTestWrapper* is indeed a subclass of *TestWrapper*.

Table II shows the lexical view computed on only two operation identifiers *updateSalomeConf* and *updateProjectConf* extracted from the same interface (namely *ISQLConfig*) of Salome-TMF. Here, the process extracted a *updateConf* concept not initially present in the identifiers. By checking the class implementing this interface (named *SQLConfig*), we remarked that these two operations call another operation, *updateConf*, defined only in the *SQLConfig* class. That clearly shows that our approach is able to create relevant new concepts from the linguistic information found in identifiers.

IV. VALIDATION

We validate the results given by our approach by running two different experiments. First, we assess the results given by the natural language processing (NLP) techniques we used in our approach (tokenization, part-of-speech (POS) tagging and dependency sorting). This experiment is given in Section IV-A. Second, we extracted lexical views from several real world softwares. We assess the quality of these views by the use of metrics. This experiment is described in Section IV-B. In Section IV-C we discuss the limits of our approach as well as its potential applications.

A. Natural language processing techniques

The goal of this experiment is to check whether the NLP techniques we used in our approach give satisfying results on identifiers from the real world. By NLP techniques, we refer to the *tokenization*, *POS tagging* and *dependency sorting* steps. For this purpose, we established a list (shown in Table III) of 24 real Java programs. In this experiment, we want to assess the efficiency of our NLP techniques on every kind of identifiers. Therefore, we drew at random from every program of our corpus 5 classes, attributes and operations identifiers. After this operation, we had 120 class identifiers, 120 attribute identifiers and 120 operations identifiers, for a total of 360 identifiers. With the help of NLP experts, we segmented manually those identifiers. Then, we manually affected parts-of-speech to the different segments. Finally, we performed the dependency sorting by hand. This set of 360 identifiers manually curated is our test corpus.

For the sake of the clarity, we split our set of 360 identifiers I in three sets $I_k, k \in (C, A, O)$. C stands for the classes, A for the attributes and O for the operations. The different I_k contain the identifiers of type k. In this experiment, we will prefix by m a function to indicate that its result has been computed manually. To show the efficiency

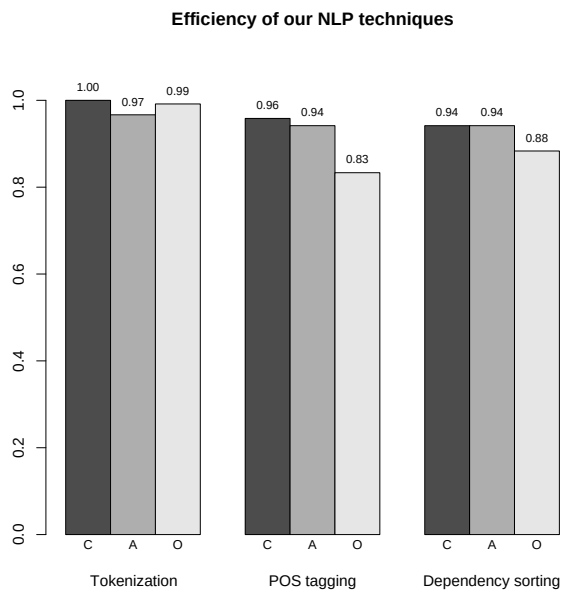


Figure 3. Efficiency of our NLP techniques

Program	Classes		Attributes		Methods	
	$\|I_c^p\|$	$\|CC_c^p\|$	$\|I_a^p\|$	$\|CC_a^p\|$	$\|I_o^p\|$	$\|CC_o^p\|$
JSON	16	14	12	12	124	35
SableCC	198	98	401	127	1711	124
JavaCC	129	69	320	153	1559	194
OpenCloud	21	10	54	36	204	31
Salome TMF	68	33	105	63	613	46
JUnit	115	65	110	53	423	96
NgramJ	37	21	63	43	136	46
JWNL	103	47	155	82	740	80
SimMetrics	114	48	118	32	490	41
Commons CLI	20	12	36	24	131	32
Args4J	29	13	25	17	98	24
JSAP	67	24	107	48	349	39
Choco	461	142	947	331	4337	352
Colt	424	150	807	324	3598	423
JGA	216	108	452	151	1425	150
JScience	176	98	320	157	1890	187
JSci	172	65	206	91	1620	78
Commons Math	223	68	383	173	1501	161
Lucene	269	91	913	303	2053	253
JCommon	181	82	332	158	965	116
XOM	323	95	268	123	1801	170
Julia	423	115	560	202	2035	183
Weka	1164	435	6013	1235	15287	976

Table III
NUMBER OF IDENTIFIERS AND CONNECTED COMPONENTS FOR THE PROGRAMS OF OUR CORPUS

of our NLP techniques, we compute the following precision metrics:

- $p^k_{tok} = \frac{|i \in I_k, tok(i) = mtok(i)|}{|I_k|}$: this metric shows the ratio of identifiers that has been successfully tokenized,
- $p^k_{tag} = \frac{|i \in I_k, tag(tok(i)) = mtag(i)|}{|I_k|}$: this metric shows the ratio of identifiers that has been affected correct parts-of-speech,
- $p^k_{dsort} = \frac{|i \in I_k, dsort(tag(tok(i))) = mdsort(i)|}{|I_k|}$: this metric shows the ratio of identifiers for which the dependency sorting has been applied correctly.

The more important metric is p^k_{dsort}, because the computation of the lexical views is based directly on the results given by this step.

Figure 3 shows the results of this experiment. It clearly shows that the NLP techniques are very efficient for the class and the attribute identifiers. But one can see a significantly lower efficiency regarding the operation identifiers. A manual assessment of this result showed that this phenomenon is induced by the POS tagging step, that sometimes fails to correctly tag tokens coming from operation identifiers. More precisely, it tends to affect parts-of-speech such as *past participle* to the action verbs of these identifiers (like *put*) whenever the past participle is the same as the present. Since we consider past participle as nouns when performing the dependency sorting (e.g. *LinkedList* is not sorted like *linkPeople*), this error results in wrongly sorted identifier tokens. Nevertheless, the overall efficiency of $dsort$ for the operation identifiers remains good (88% of the identifier tokens well sorted).

B. Lexical views

In this second experiment, we build the lexical views corresponding to the different programs of our corpus. For each program we compute three lexical views lv_k: the *class lexical view*, the *attribute lexical view* and the *operation lexical view*. The goal of this experiment is to show that our approach successfully discovers several groups of semantically linked identifiers among the different identifiers of a program. In a lexical view, such a group of semantically comparable concepts is a connected component. Therefore, the more the different identifiers are *semantically* linked, the less is the number of connected components in the lexical view. In order to show that, we compute six metrics. We remind that I_k is the set of identifiers of kind $k \in (C, A, O)$. Therefore, for every program, I_c is the number of classes, I_a the number of attributes and I_o the number of operations. We denote by CC_k the set of connected components in lv_k. Table III shows the results of this experiment.

In Table III, we can see that when there are only a few identifiers of a kind, the number of connected components is similar to the number of identifiers. On the other hand, when there is a large amount of identifiers, the number of connected components is dramatically lower than the number of identifiers. To show the differences between the number of identifiers and connected components for every kind of software elements, we compute several additional metrics. We count ai_k the average number of identifiers of kind k in a program, and acc_k the number of connected components of kind k in a program. Figure 4 shows ai_k and acc_k for $k \in (C, A, O)$. This figure clearly shows that the reduction is similar for the classes and the attributes, while

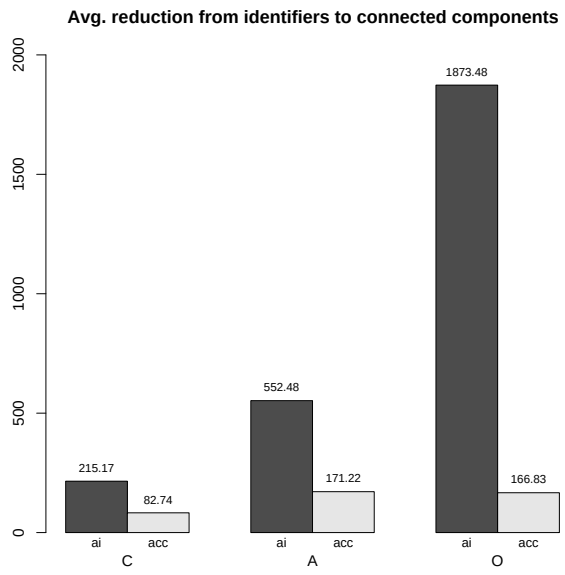


Figure 4. Average reduction from identifiers to connected components

it is significantly higher for the operations. This results was expected because the actions performed by the operations are highly similar in a software, but applies on different objects. Indeed, softwares usually contain a lot of operations prefixed by an action verb such as *get*, *set* or *update*. The operations are thus grouped in connected components corresponding to these action verbs. For this reason, the reduction on the operation identifiers is higher than with the other identifiers. In an operation lexical view, it could therefore be very interesting to focus also on the second level concepts.

With the previous metrics, we assessed if our lexical views were composed of connected components. To improve the understanding of the lexical view we produce, we compute two additional metrics. Firstly, let LV_k be the set of lexical views of identifiers of kind k. We remind that CC_k is the set of connected components in $lv_k \in LV_k$. We compute ad_k the average depth of $cc_k \in CC_k, |cc_k| > 1$. Similarly we calculate as_k the average size of $cc_k \in CC_k, |cc_k| > 1$. Finally, we have an_k the average number of new concepts (therefore extracted during the lexical expansion step) $cc_k \in CC_k, |cc_k| > 1$. To sum up, ad_k represents the average depth of the connected components coming from identifiers of kind k and of size greater than 1. Similarly, as_k represents the average size of these connected components, and an_k the average number of new concepts. Figure 5 shows these metrics. We see that the average depth is similar for every kind of identifiers. An average maximum depth about 2 means that, in average, the connected components have three levels of hierarchy. It is not very deep compared to WordNet. The average size of the connected components differs according to the kind of the considered identifiers. First, the attribute identifiers yield the smaller connected components. Second, classes lead to slightly larger connected components. Finally, operations result in significantly larger connected components. The size of the operation connected components is probably induced by the phenomenon previously described, but it seems that class identifiers yield bigger connected components than the attributes. Finally we can see that operations induce the greatest number of new concepts (which are the different action verbs). Classes yield a fair number of new concepts. In general the concepts deduced from the classes are very useful to understand the program. Finally, attributes seem to produce only a small amount of new concepts.

C. Discussion

In this section, we discuss several limitations and possible improvements of our approach. Some of those have also been observed in a related approach [9].

Naming conventions During the validation of our approach, we used Java programs. The naming conventions used in Java make it easy to tokenize the identifiers. Nevetheless, it remains several ambigous cases when camel case is used. Our approach would therefore benefit from [10] where identifiers are segmented using a corpus of software, allowing to dectect the most likely tokenization. On the other hand, some other naming conventions, specially in legacy code, yield identifiers not so easy to tokenize. It is even possible that no tokenization at all has been put in the identifiers, leading to an impossible tokenization with a classic technique. In this case, other dedicated approaches could be used [11].

POS tagging We compute the parts-of-speech of the tokenized terms by using a version of Tree Tagger that has been trained for the general English language. We saw in the validation that it yields wrong results in several cases. Nevertheless, a version of tree tagger specifically trained to tag terms coming from identifiers would achieve better results. Training Tree Tagger requires to create a large corpus of sentences manually tagged.

Generated identifiers It is not uncommon to find programs where several identifiers have been generated. These identifiers are most of the time meaningless and will therefore introduce some noise in our lexical views. Nevertheless, if they are generated following strict rules (like they are all prefixed by the same term), it is very likely that they have been all organized in the same connected component of the lexical view, and therefore it might be possible to easily filter them from the lexical view.

Abbreviations and synonyms In real world identifiers, it is common that both expanded and abbreviated forms of a term are used. For instance, let us consider a *Message* class that has a huge amount of subclasses. These subclasses

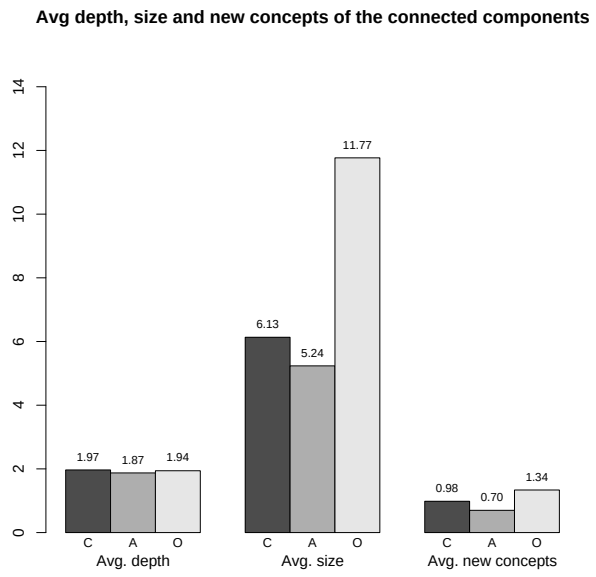


Figure 5. Average size, depth and new concepts in the connected components

would probably be named *ErrorMsg*, *WarningMsg*, in order to keep the identifiers short. With our approach, those identifiers would be in two different connected components of the lexical view. However, it is easy to extend our approach to cope with this case. Only a little modification of the lcp function described in Section III-D is required, by extending the $=$ relation between the terms to look in a user-defined list of abbreviations. This extension can also be applied to deal with the case where two synonym words are used in a software (e.g. *car* and *auto*). Moreover, it would possible to couple our approach with the appraoch in [10], that discover automatically abbreviated forms within a software source code.

General purpose ontologies We said in Section I that general purpose ontologies, like WordNet, are difficult to use with the identifiers. Nevertheless, they still could bring valuable information to our lexical views. Since our lexical views are ordered with a $hypo$ relation, trying to extract and inject information from WordNet to our lexical view is easier than with the raw identifiers. Indeed, it is sufficient to look to the more general nodes of our lexical views and try to connect them (semi-automatically) with the nodes from WordNet.

Semantic measures WordNet has been widely used as the basis to define and be able to compute *semantic* similarity measure between the words [4]. Therefore, it is possible to take advantage of this work to compute semantic similarity measures between the identifiers.

V. Related work

Automatic extraction of lexical relations between words has been extensively in the field of Natural Language Processing (example: [12], [13]). Usually the existing approaches relies on syntactical clues relative to a given language (for instance, in English, *such as* indicates often a case of hyponymy). Unfortunately, these approaches cannot be applied on identifiers since the syntactical clues are removed in a identifier.

Perhaps the most related approach to our is the one described in [14]. In this approach a process similar to our is defined to extract what the author call *verb - direct object* pairs (like *draw - circle*). The authors explain how their approach can be used to locate concerns or to mine aspects. Our approach differs from this one especially on the output: we focus on a WordNet-like identifier network that shows the lexical relations between the identifiers of a software.

In [10], an approach that automatically mines abbreviation from a software source code is described, by using pattern matching techniques. In [15] an automatic approach that uses the word frequencies in a software source code to tokenize the identifiers, allowing to tokenize identifiers with ambiguous camel case or with no syntactical clue at all. Although these approaches pursue different objectives than our, our approach would probably greatly benefit from using them.

In [16], six approaches that computes the semantic similarity of words using WordNet are compared on word coming from identifiers. The article concludes that WordNet seems not adapted to this task, since synonyms in the general English are not the same as in software identifiers. This indicates that our approach, that automatically extract a WordNet of identifiers, might improve computation of semantic similarity on software identifiers.

In [17] a manual procedure extracts words and constructs a dictionary including the words and their part-of-speech categories in identifiers. This dictionary is manually enriched during an identifier segmentation phase, where successive suffixes are looked for in the dictionary. Words are also assigned a class among 7 lexical classes including number, acronym or english-form. Then a grammar for the language of identifiers is defined, which explains how identifiers are built. A concept lattice is used to group identifiers that share common words. An analysis of identifiers of 9 systems shows that most of the words are English-forms. The concept lattice highlights relevant terms in identifiers and recurrent grouping, revealing important concepts.

A technique that aims at extracting concepts from file names is proposed in [11]. The final objective is clustering files into subsystems. The method is composed of two steps. First step constructs candidate segments. Two directions are followed: an iterative method uses n-gram decomposition of words and an English dictionary, and removes

prefixes candidates to find other candidates; a statistical method computes most common substrings in file names. Comments, function identifiers are also explored to find candidate segments and abbreviations are computed for the candidates. The second step uses the candidate segments and abbreviations to split file names. Several strategies are explored and their combination gives about 90% of correct decompositions. Our approach, although already giving result, could be improved by some of their proposals.

Several approaches [18], [9] use Latent Semantic Indexing (LSI), a well known information retrieval technique, to find cluster in softwares. Software artifacts are modeled as documents whose text content are the word contained in its identifier and plain-text comments. Then the LSI technique is applied, making it possible to compute the similarity between two software artifacts. These similarity values are then used to find clusters of similar software artifacts. Our approach is different because it focus on extracting the hierarchical relations between the identifiers. This cannot be done by using LSI.

VI. Conclusion

The names of the identifiers are recognized to contain a large part of the semantics of programs, and thus, if adequately chosen, to be of great help to understand programs.

The contribution presented in this paper is a fully automated approach that extracts from those identifier names the main concepts of a program, and that organizes them into a lexical view making explicit the relations linking them. The main benefits of this approach are:

- it is fully automated,
- it is language- and paradigm-independent,
- it can be applied to any piece of code, of any granularity, from a single class to an entire system, in order to analyze any kind of identifiers (class names, function names, attributes names),
- it extracts the concepts explicitely included in the identifiers, as well as those implicitely included,
- it organizes the concepts in a lexical view allowing to navigate the concepts through their hierarchical structure.

The validity and interest of the approach have been demonstrated with a case study involving 24 existing Java programs.

In order to improve the approach, we plan to create a training set in order to train a POS tagger specialized in tagging english sentences derived from identifiers. We also plan to slightly modify the approach to let the user give synonymy information.

Linguistic information in the form of lexical views may be helpful in several phases of software lifecycle, starting from assistance for identifier naming, to maintenance and reengineering. Identifier analysis is for example a crucial part of several approaches allowing class models to be improved, using automatic detection of design defects [19], class model restructuring based on the entity names [20], object identification based on synonymous files [21] or aspect mining approaches that use fragments of identifiers [22], group programming elements according to relevant substrings of their names [23] or search for lexical and type-based patterns () [24]. Our approach is currently used in a class model restructuring technique [25], and we will investigate the way it can contribute to enhance other approaches requiring identifiers analysis like in references [19], [22], [23], [24].

Acknowledgment

The authors would like to thank France Télécom R&D for their support of this work (CPRE 5326).

References

[1] A. Abran, P. Bourque, R. Dupuis, and L. Tripp, "Guide to the software engineering body of knowledge (ironman version)," Technical report, IEEE Computer Society, Tech. Rep., 2004.

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

[3] G. A. Miller, "Wordnet: A Lexical Database for English," in *HLT*. Morgan Kaufmann, 1994.

[4] A. Budanitsky and G. Hirst, "Evaluating WordNet-based Measures of Lexical Semantic Relatedness," *Computational Linguistics*, vol. 32, no. 1, pp. 13–47, 2006.

[5] A. Aho, M. Garey, and J. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, p. 131, 1972.

[6] H. Schmid, "Probabilistic Part-of-Speech Tagging Using Decision Trees," in *Proceedings of the International Conference on New Methods in Language Processing*, Manchester, UK, 1994, pp. 44–49.

[7] L. Tesnière, *Eléments de syntaxe structurale*. Paris: Klincksieck, 1959.

[8] I. A. Mel'čuk, *Dependency Syntax: Theory and Practice*. New York: State University of New York Press, 1988.

[9] A. Kuhn, S. Ducasse, and T. Gîrba, "Enriching Reverse Engineering with Semantic Clustering," in *proc. of the 12th Working Conference on Reverse Engineering (WCRE 2005)*. IEEE Computer Society, 2005, pp. 133–142.

[10] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. L. Pollock, and K. Vijay-Shanker, "Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *MSR*, A. E. Hassan, M. Lanza, and M. W. Godfrey, Eds. ACM, 2008, pp. 79–88.

[11] N. Anquetil and T. Lethbridge, "Extracting concepts from file names: A new file clustering criterion," in *ICSE*, 1998, pp. 84–93.

[12] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *COLING*, 1992, pp. 539–545.

[13] P. Pantel and M. Pennacchiotti, "Espresso: Leveraging generic patterns for automatically harvesting semantic relations," in *ACL*. The Association for Computer Linguistics, 2006.

[14] Z. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker, "Analysing source code: looking for useful verb–direct object pairs in all the right places," *Software, IET*, vol. 2, no. 1, pp. 27–36, 2008.

[15] G. Sridhara, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *ICPC*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 123–132.

[16] ——, "Identifying word relations in software: A comparative study of semantic similarity tools," in *ICPC*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 123–132.

[17] B. Caprile and P. Tonella, "Nomen Est Omen: Analyzing the Language of Function Identifiers," in *proc. of the Working Conference on Reverse Engineering (WCRE'99)*, 1999, pp. 112–122.

[18] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *proc of the 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE Computer Society, 2004, pp. 214–223.

[19] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien, "A domain analysis to specify design defects and generate detection algorithms," in *FASE*, ser. Lecture Notes in Computer Science, J. L. Fiadeiro and P. Inverardi, Eds., vol. 4961. Springer, 2008, pp. 276–291.

[20] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau, "Design of Class Hierarchies Based on Concept (Galois) Lattices," *TAPOS*, vol. 4, no. 2, pp. 117–134, 1998.

[21] A. Cimitile, A. D. Lucia, G. A. D. Lucca, and A. R. Fasolino, "Identifying objects in legacy systems using design metrics," *Journal of Systems and Software*, vol. 44, no. 3, pp. 199–211, 1999.

[22] W. G. Griswold, Y. Kato, and J. J. Yuan, "Aspect Browser: Tool Support for Managing Dispersed Aspects," in *In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems - OOPSLA 99*, 1999.

[23] T. Tourwé and K. Mens, "Mining Aspectual Views using Formal Concept Analysis," in *SCAM*. IEEE Computer Society, 2004, pp. 97–106.

[24] C. Zhang and H.-A. Jacobsen, "PRISM is research in aSpect mining," in *OOPSLA Companion*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 20–21.

[25] G. Arévalo, J.-R. Falleri, M. Huchard, and C. Nebut, "Building abstractions in class models: Formal concept analysis in a model-driven approach," in *MoDELS*. Springer, 2006, pp. 513–527.

[26] R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds., *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*. IEEE Computer Society, 2008.