Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources

Kalina Bontcheva[1] and Marta Sabou[2]

[1] Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield, UK
kalina@dcs.shef.ac.uk
[2] Knowledge Media Institute (KMi), The Open University
Milton Keynes, UK
R.M.Sabou@open.ac.uk

Abstract. While early efforts on applying Semantic Web technologies to solve software engineering related problems show promising results, the very basic process of augmenting software artifacts with their semantic representations is still an open issue. Indeed, existing techniques to learn ontologies that describe the domain of a certain software project either 1) explore only one information source associated to this project or 2) employ supervised and domain specific techniques. In this paper we present an ontology learning approach that 1) exploits a range of information sources associated with software projects and 2) relies on techniques that are portable across application domains.

1 Introduction

Large software frameworks and applications tend to have a significant learning curve both for new developers working on system extensions and for other software engineers who wish to integrate relevant parts into their own applications. This problem is made worse in the case of open-source projects, where developers are distributed geographically and also tend to have limited time for answering user support requests and generally helping novice users by writing extensive tutorials and step-by-step guides. At the same time, such online communities typically create a large amount of information about the project (e.g., forum discussions, bug resolutions) which is weakly integrated and hard to explore [1]. In other words, there is a practical need for better tools to help "guide users through the jungle of APIs, tools and platforms" [12].

Recent research has begun to demonstrate that Semantic technologies are a promising way to address some of these problems [12]. For instance, search and browse access to web service repositories can be improved by a combination of ontology learning and semantic-based access (e.g., ontology-based browsing and visualisation) [15]. Similarly, semantic-based wikis have been proposed as a way of supporting software reuse across projects, where a domain ontology describing the different software artifacts emerges as a side effect of authoring wiki pages [7]. Finally, the Dhruv system [1] relies on a semantic model obtained by

instantiating hand built domain ontologies with data from different information sources associated with an open source project to support bug resolution.

Because ontologies are core to the Semantic Web, a prerequisite for all the above approaches is the existence of a domain ontology that describes the given software artifact. However, this is seldom the case and, as a result, the task of automatically building such ontologies becomes very important.

The goal of Ontology Learning (OL) methods is to derive automatically (parts of) an ontology from existing data (for state of the art overviews see [3]). In earlier work it has been demonstrated that textual sources attached to software artifacts (e.g., software documentation) contain a wealth of domain knowledge that can be automatically extracted to build a domain ontology [14]. It was observed however that existing generic ontology learning approaches need to be adapted to handle the particularities of software specific textual sources (i.e., low grammatical quality, using a sublanguage). The approach presented in [14] explores the particularities of the sublanguage specific to software documentation to manually derive knowledge extraction rules. While the most generic rules can be used across domains they only extract a limited amount of knowledge. More specific rules need to be manually identified for new domains. Also, this technique has been applied on a single type of software specific textual sources, namely short functionality descriptions. This is a drawback because the work of Ankolekar et al. [1] showed that knowledge about a software project is often spread in several different information sources such as source code, discussion messages, bug descriptions, documentation and manuals.

Taking into consideration the lessons learnt from previous approaches, in this paper we present an approach to learning domain ontologies from *multiple* sources associated with the software project, i.e., software code, user guide, and discussion forums. Our technique does not simply deal with these different types of sources, but it goes one step further and exploits the redundancy of information to obtain better results. The strength of our technique is that it relies on unsupervised learning methods that, unlike earlier work, are portable across domains. These were successfully used to identify domain concepts. Because the system is currently being developed we only report on this domain concept identification process which is a basic step in any ontology learning task.

In the next section we present the particularities of a case study in which we apply our technique. In Section 3 we briefly describe the characteristics of three major data sources that we use as basis for ontology learning. In Section 4 we detail our the concept identification aspect of the ontology learning approach and present experimental results in Section 5. We conclude with a final discussion and future work description (Sections 6 and 7).

2 GATE: A Case Study

GATE[3] [6] is a world-leading open-source architecture and infrastructure for the building and deployment of Human Language Technology applications, used

[3] http://gate.ac.uk

by thousands of users at hundreds of sites. The development team consists at present of over 15 people, but over the years more than 30 people have been involved in the project. As such, this software product exhibits all the specific problems that large software architectures encounter and has been chosen as a case study in the context of the TAO[4] EU-funded project.

While GATE has increasingly facilitated the development of knowledge-based applications with semantic features (e.g. [2, 11, 14]), its own implementation has continued to be based on compositions of functionalities justified on the syntactic level, understood by informal human-readable documentation. By its very nature as a successful and accepted 'general architecture', a systematic understanding of its concepts and their relation is shared between its human users. It is simply that this understanding has not been formalised into a description that can be reasoned about by machines or made easier to access by new users. Indeed, GATE users who want to learn about the system are finding it difficult due to the large amount of heterogeneous information, which cannot be accessed via a unified interface.

The advantage of transitioning GATE to ontologies (i.e., describing its resources with ontology based annotations) will be two-fold. Firstly, GATE components and services will be easier to discover and integrate within other applications (see [15, 16]). Secondly, users will be able to find easily all information relevant to a given GATE concept, using concept-based search on the GATE documentation, XML configuration files, video tutorials, screen shots, user discussion forum, etc.

A concrete example of the benefits of using semantic technologies to manage access to knowledge about GATE relates to facilitating access to discussion forums. Indeed, discussion forums are continuously updated with new postings, so the main challenge comes from implementing a process which indexes them every day with respect to a domain ontology. For instance, GATE's discussion forum has on average about 120 posts per month, with up to 15 on some days. Due to the volume of this information, it would be helpful if developers could choose to read only postings related to their areas of interest. Therefore, what is required is automatic classification of postings with respect to concepts in the ontology and a suitable interface for semantic-based search and browse. A similar problem is being currently addressed in the context of digital libraries [17] and we will consider using some of these techniques as well.

Since users tend to post messages on the discussion forums when they have failed to identify a solution to their problem in the user manuals and earlier forum postings, by analysing also which topics are being discussed one can also identify potential weaknesses in the current documentation, which can then be rectified. Again this is an example, where classification with respect to an ontology can help with the maintenance and update process of software documentation.

[4] http://www.tao-project.eu

3 Software Artefacts as Data Sources for Ontology Learning

In general, present day software development practises lead to the creation of multiple artifacts, which implicitly contain information from which domain ontologies can be learnt automatically. These multiple data sources can be classified along two orthogonal dimensions:

Structured vs. unstructured: Source code, WSDL files, XML configuration files, and log files are all examples of structured artifacts. Web pages, manuals, papers, video tutorials, discussion forum postings, and source code comments are all unstructured. Due to this diverse structure and content the challenge here is how to choose and customise the ontology learning methods, so that they can achieve the best possible results with minimum human intervention. Another aspect that is worth considering here is whether some knowledge is easier to acquire from only some of these sources (e.g., key terms from the source code comments), and then combine this newly acquired knowledge with information from the other sources (for an application of this approach in multimedia indexing see [9]).

Static vs. dynamic: As software tends to go through versions or releases, i.e., evolve over time, the majority of software-related data sources tend to change over time, albeit some more frequently than others. For example, API and web service definitions, configuration files, manuals, and code comments would be relatively stable between major releases, whereas discussion forum postings would change on a daily basis. In fact, it is the dynamic nature of software data sources which poses a particular challenge, as the ontology learning methods would need to be made sensitive to the changeable nature of the domain which they are trying to capture. In particular, methods for ontology versioning and evolution would be required [10]. This also includes modelling of temporal aspects, e.g., date from which a concept was introduced or a comment was made.

In addition, each data source has its own specific characteristics, which need to be taken into account.

3.1 Source code

Learning domain ontologies from source code poses several challenges. Firstly, each programming language and software project tends to have naming conventions and these need to be considered. In a nutshell, the goal is to *separate variable and method names* into their constituent words, i.e., `getDocumentName` should be separated into *get*, *document*, and *name*, prior to being submitted as input to the ontology learning algorithms.

The second problem is that the ontology learning methods need to distinguish between terms specific for the programming language being used (e.g., *hash maps* for Java) and the application-specific terms, i.e., the terms which are relevant

to the ontology (*document names* in the case of GATE). This problem has also been recognised by Ankolekar et al. which distinguishes between *code terms* that denote programming language specific elements and *noun phrases* that stand for domain specific terms [1].

Finally, many of the extracted terms can refer to the same concept. A simple example is considering lexical variants of a term as pointing to the same concept, such as singular and plural forms of nouns (e.g., documents and document) and different forms of a verb (e.g., return and returning). A more complex case is that when syntactically different terms refer to the same concept or instance. One example from the GATE system is the part-of-speech tagger, which is also referred to as POS tagger and Hepple tagger.

3.2 Software Manuals

Software projects typically have at least a user manual, but bigger ones would also have a programmers' guide and an installation manual. These are all unstructured data sources which can also come in different formats, e.g., PDF, Word, HTML. Therefore, in the first instance, we need to be able to read these formats and extract the text content from them.

Due to their size (some over hundreds of pages) and their lack of structure, it is our view that manuals are more suitable for extracting hierarchical and other relations between concepts, but not so suitable for learning the concepts in the first instance. Previous work on ontology learning [5] has indeed demonstrated that large amounts of unstructured text can be used successfully to learn subsumption between two concepts A and B, using Hearst-like lexical patterns such as `<conceptA> isa a <conceptB>`.

3.3 Discussion Forums

From a content analysis perspective, forums present a challenge as they are unstructured but also they require special format analysis techniques, in order to identify which thread a posting belongs to (or a new thread) and where in the message body there is quoted text, if any.

The problem with identifying different terms which refer to the same concept in the ontology arises even more strongly here, as some inexperienced users might not be using the correct terminology. For instance, in GATE there is a component called noun phrase or an NP chunker, but in some posts it is being referred somewhat incorrectly as noun chunker.

4 Multi-Source Ontology Learning

As we stated in the introduction, learning ontologies from software artifacts is an important task that has already been pioneered in the context of Web services [14]. However, while this early work demonstrates the feasibility of the idea that important knowledge can be extracted from software artifacts, it falls short

of taking into account the nature of such artifacts. Indeed, large scale software projects produce large, distributed, heterogeneous and dynamically changing information sources. Our view is that a logical next step is to provide ontology learning methods that can explore the wealth of knowledge provided by a range of information sources typically associated with software projects. Based on our analysis of some typical data sources (see Section 3) and lessons learnt from previous work [14], we identified the following requirements for ontology learning methods that explore software artifacts:

- ability to deal with large and distributed document collections
- operate on a dynamically growing document base
- cover heterogeneous data sources
- benefit from redundancy of information from multiple sources
- deal with ontologies evolving over time, e.g., new concepts appearing
- maintenance of different versions of the ontology, corresponding to different versions of the software

In this section we describe an ontology learning method that, for now, addresses the requirements raised by taking into account multiple data sources. We describe the concept extraction process which benefits from combining information from different information sources. At present, we are in the process of extending the system towards learning relations and the initial experiments have proved promising (see Section 7).

4.1 System Overview

Our multi-source ontology learning system uses the language processing facilities provided by GATE itself [6, 2, 9] and we have modified or extended some of them specifically for the problem of learning from software artifacts. Note that GATE plays a dual role in our research – both as one of the software projects used for experimenting with our technology and also as the language processing software infrastructure, which we used for building the technology itself. Overall the process consists of four main stages, shown on Figure 1:

Term extraction from source code. An important lesson from the ontology learning research is that there are no generic ("one-size-fits-all") methods and therefore any OL method will need some degree of adaptation to a new domain - or even to new data sets within the same domain. As a result, we employ different learning methods to deal with different data sources. The left hand side of the figure depicts the steps needed to process structured sources such as source code, while the right hand side deals with unstructured sources such as manuals and forum discussions. Relevant terms that are extracted from source code (Section 4.2) are pruned (Section 4.3) to exclude irrelevant hits.

Term extraction from documentation and forums. The key domain terms identified in the source code are used as a starting point for exploring less

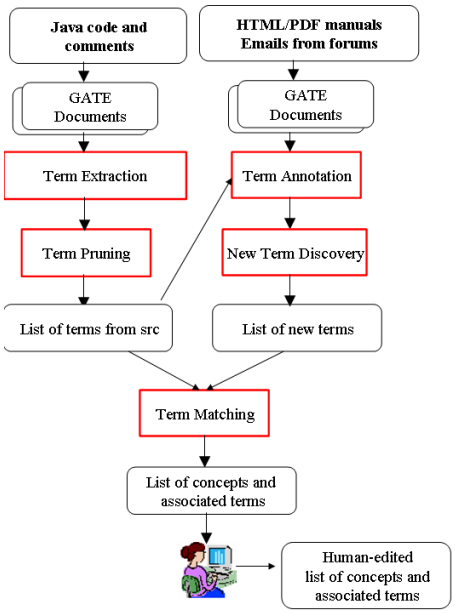

Fig. 1. Multi-source Ontology Learning from Software Artefacts

structured sources (Section 4.4). First, the location of these terms is identified in the textual sources (Term Annotation) and then new terms are discovered taking into account these annotations.

Concept identification. The terms discovered both in the source code and in other textual sources are merged and submitted to a concept identification process (Term Matching) that identifies terms referring to the same concept (Section 4.5). The identified concepts are passed on to a user for validation.

User validation. An important point to make is that the automated methods are not intended to extract the perfect ontology, they only offer support to domain experts in acquiring this knowledge. This help is especially useful in situations like ours when the knowledge is distributed in several documents (possibly of different types: text, diagrams, video, etc). In fact no existing OL technique is completely unsupervised: a domain expert must be included somewhere in the knowledge acquisition loop. Therefore, the automatically acquired knowledge is post-edited, using an existing ontology editor, to remove irrelevant concepts and add missed ones. The link between parts of the content where learnt concepts occur and the concept itself are preserved in the ontology, in order to enable the domain experts to examine the empirical grounding of the ontology into the software artifacts.

In the remaining of this section we present the details of the first three of these extraction stages.

4.2 Extracting Terms from Source Code

In order to identify the key terms, specific to the given software project, we chose to analyse the source code and its accompanying comments, because of their semi-structured nature. The term extraction process consists of three components, all implemented within GATE.

The first step is to deal with code naming conventions (see Section 3.1), we implemented a special source code tokeniser, which is based on the default GATE English tokeniser but is capable of separating class and variable names into their components, e.g., *VisualResource* into *Visual* and *Resource*. The example is shown on Figure 2, where all tokens are marked in blue and the Token pop-up window shows the information for a selected token (e.g., Resource).

Next in the pipeline is the English morphological analyser, which is being used to annotate all words with their root forms (e.g., the root of the word "resources" is "resource"). The goal is to derive the same term from the singular and plural forms, instead of two different terms.

The third component is the GATE key phrase extractor [9], which is based on TF.IDF (term frequency/inverted document frequency). This method looks for phrases that occur more frequently in the text under consideration than they do in language as a whole. In other words, TF.IDF finds phrases that are characteristic of the given text, while ignoring phrases that occur frequently in the text simply because they are common in the language as a whole. It requires training data in order to determine how common each phrase is, but this training data need not be marked up with any human-created annotations.

When TF.IDF is applied to source code, we need a training corpus of source code of another system, implemented in the same programming language. In this way, high frequency terms specific to the programming language can be eliminated and only terms specific to the given software project would be selected as relevant to the ontology. For instance, terms such as *while*, *hash map*, and *list* are Java terms and are thus not relevant. Therefore, we first trained the TF.IDF extractor on the source code of the Sesame[5] open source project, because it is implemented in Java and is of a comparable size to the GATE code base. Then, given the frequency statistics obtained from the Sesame code, we extract up to ten key terms from each GATE java file. For example, Figure 2 shows the code of VisualResource.java, with the extracted terms annotated in brown (e.g., visual resource, GUI, GATE, resource).

The difference between this approach and the rule-based approach for concept identification (e.g., [14]) is that this is an unsupervised learning method which makes it porting across different data sources and application domains a lot more straightforward. In effect, all that is required is a collection of unannotated texts, which are similar in structure to the data sources which need to be analysed.

[5] http://www.openrdf.org/

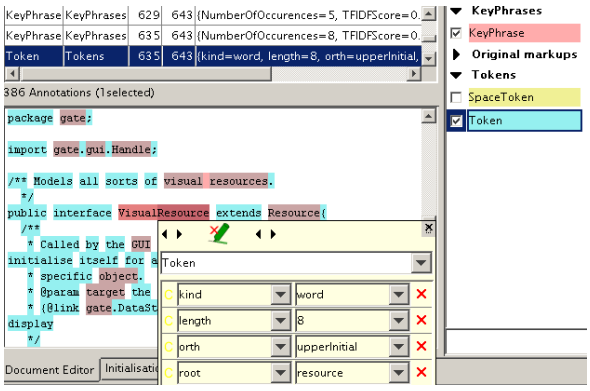

Fig. 2. Example token and key term annotations

4.3 Term Pruning

The next stage in the concept learning process is to prune the list of terms in order to filter out irrelevant concepts. Previous work on learning web service domain ontologies has used average term frequency as the threshold below which concepts should be pruned [14].

In our case, the TF.IDF score associated to each term by the term extractor can be used as a basis for pruning. In the current experiments we experimented with several fixed thresholds and decided on retaining only the top three terms per source file. Another possibility, to be explored in the future, is to identify the average TF.IDF score across the entire corpus and then prune all terms with lower score, regardless of which source file they come from.

At present, based on the top three key terms from each Java file, a list of all terms from all documents is compiled and this list is considered to contain some of the labels used to refer to concepts in the domain. In other words, our approach assumes that the developers are most likely to use more than one term for referring to the same concept and therefore we should not map directly the linguistic expressions into concepts. Instead, a new processing stage needs to be introduced into the concept learning process which attempts to discover all equivalent terms which lexicalise the same concept. We call this process term matching and it is discussed in Section 4.5.

4.4 Multi-Source Term Enrichment

Term extraction from source code in itself is not sufficient, because our analysis of the results showed that the comments and class names tend to use consistently only one term to refer to each of the domain concepts. In addition, the software developers usually write the comments to be as short as possible and thus introduce abbreviations, e.g., doc for document or splitter for sentence splitter. The consequence is that the majority of terms extracted from the source code

are single word terms, whereas the user-oriented unstructured texts tend to use more multi-word terms to refer to the same concepts.

Consequently, the goal of the multi-source term enrichment process (see right side of Figure 1) is to use the unstructured software artifacts (e.g., forum postings and manuals) in order to identify new frequently occurring multi-word terms, which consist of two or more of the terms extracted from the source code.

The first step is *term annotation*, which is implemented using the GATE gazetteer component (see [6]). The gazetteer takes as an input the term list extracted from the source code and annotates (i.e., identifies and marks-up) all mentions of these terms in the unstructured documents, i.e., forum postings and manuals in our case. The annotation process is done on the root forms (i.e., the morphological information), because the list of terms contains only basic forms (e.g., *document* but not *documents*).

The second step is what we call *discovery of multi-word terms* and this is implemented as a simple regular expression, which finds two or more consecutive terms and joins them together in a multi-word term. For example, some terms derived from the source code are *ANNIE*, *sentence*, *splitter*, and *gazetteer*. When the term enrichment is run on the GATE forum postings, then several new multi-word terms are identified based on the co-occurrence of the simple terms: *sentence splitter*, *ANNIE sentence splitter*, and *ANNIE gazetteer*. The result is a list of new multi-word terms which are then merged with those extracted from the source code prior to term matching.

4.5 Term Matching

The term matching module uses a set of orthographic rules to determine terms that refer to the same concept (e.g., POS tagger and part-of-speech tagger). This component is based on the GATE Orthographic co-reference module, which identifies co-referring names in documents, e.g., George Bush and President Bush [6]. The rules that we apply for matching the terms are generally applicable (i.e., not software or domain specific) rules such as:

- *exact match*
- *equivalent*, as defined in a synonym list: this rule is used to handle matching of terms like *Hepple tagger* and *POS tagger*.
- *spurious*, as defined in a list of spurious names. This is a list, similar in structure to the list of equivalence terms, where the user can add pairs of terms which should never be matched.
- *acronyms*: handles acronyms, e.g., *part-of-speech tagger* and *POS tagger*.
- *abbreviations*: identifies whether one term is an abbreviation of another, e.g., *doc* and *document*. However, this rule would not match multi-token terms such as *tokeniser* and *Unicode tokeniser* as they are considered different.

The results of the term matching is considered to be the set of learnt concepts. We generate an OWL class with name starting with `GATE_` followed by the term (e.g., `GATE_DOCUMENT`) and assign the term as the alias of that concept.

Term	Freq.	Term	Freq.	Term	Freq.	Term	Freq.
GATE	218	test	20	licence	12	annotation set	9
annotation	63	word	18	persistence	12	creole	9
feature	41	synset	18	gazetteer	12	node	9
corpus	29	annot	18	sense	12	transducer	8
1.1	26	box	14	document	10	data store	7
doc	25	controller	13	PR	10	LR	7

Table 1. Top frequency terms extracted from the GATE source code

Where several terms are matched as equivalent, the first one is used to derive the class name (again, using the GATE prefix) and all terms are assigned as aliases (e.g., concept `GATE_POS_TAGGER` with aliases "POS tagger" and "part-of-speech" tagger).

5 Experimental Results

We experimented with performing term extraction from the 536 java source files, which constitute GATE version 3.1. The resulting term list contained 576 terms, but only 218 of them had frequency of more than 1.

Table 1 shows the top frequency terms, extracted from the GATE code. As can be seen there are only a few spurious ones. "1.1" is a version number and its inclusion can easily be avoided in future by preventing the inclusion of numbers and full stops in the terms. "Test" features so prominently because of the unit testing code in GATE, but again, this is easily rectified by allowing the user to exclude some source files from processing. "Licence" is included because we did not exclude the copyright notices from the analysis.

More interestingly, in the case of two important concepts - documents and annotations - we extracted two representative terms one of which is an abbreviation (i.e., "doc" and "annot"). As already discussed above, such abbreviations are frequently used in source code when the concept names are too long.

The 576 terms were then given as input to the term enrichment process, which (due to time constraints with the paper deadline) we ran on only 2000 of the GATE forum postings, posted between January 2005 and June 2006. The term enrichment produced 153 multi-word terms, with only 12 terms overlap with the source code term list. Table 2 shows the top frequency multi-word terms. They all denote important concepts in GATE. Only two of them have more general software relevance ("xml file" and "jar file"), whereas all others are GATE-specific (e.g., JAPE is the name of the most popular GATE component – its pattern-matching engine, hence the files, which contains grammars consisting of rules).

Overall, when the two lists were combined, we obtained 719 terms in total, 286 of which had frequency more than 1. These 286 terms were then used for a limited evaluation, carried out by an expert GATE developer who counted all

Term	Freq.	Term	Freq.	Term	Freq.	Term	Freq.
pos tagger	44	jape rule	33	property name	27	token annotation	23
jape file	34	sentence splitter	30	jape transducer	24	jar file	19
jape grammar	34	gazetteer list	27	xml file	24	creole plugin	15

Table 2. Top frequency terms extracted from GATE forum postings

errors. They identified 76 spurious terms, which gives precision of 73.4%. An evaluation of the method's recall is forthcoming (see Section 7).

This also brings us to the more general problem of user validation in the ontology learning process. On one hand it should be kept to a minimum by making our methods as accurate as possible, but on the other, this should not come at the expense of recall. Therefore, as discussed in Section 4.1, ultimately the set of learnt concepts will be verified by a domain expert, who would use an ontology editor in order to delete wrongly identified concepts or add ones missed by the automatic learning process.

6 Discussion

This paper presented our initial prototype of an ontology learning system, created in the wider context of using semantic technology to facilitate access, maintenance and re-use of software artifacts, produced by large open source projects.

The first characteristic and strength of our technique is that it deals with multiple information sources. Indeed, the system contains different steps for dealing with structured data sources on one hand (i.e., source code) and unstructured sources on the other (i.e., documentation). Note that the separate extraction steps do not simply run in parallel but instead are integrated in such a way that the performance of the method is optimised: simple terms are extracted from the source code and then used as a starting point for identifying compound terms in the user documentation.

It should also be noted that our approach addresses most of the particularities of the used data sources described in Section 3. We use an English tokeniser to deal with naming conventions and a reference corpus consisting of the source code of Sesame to filter out code terms. By using GATE and its built in document management functionalities we can easily process and read the various different formats. Finally, we rely on an a morphological analyser and a co-reference module to identify multiple localisations of the same concept and therefore obtain a list of concepts from a much larger set of identified relevant terms.

The second strength of our approach with respect to what has been done in the field of learning ontologies from software artifacts is that it relies on an unsupervised method to identify domain terms by comparing the processed corpus with a reference corpus. While similar techniques have been employed in previous, generic ontology learning approaches such as [4] and [13], our experiments suggest that they can be successfully applied also in the software engineering

context. In particular, the challenge we address here is not so much the development of novel ontology learning methods, but rather the improvement of their robustness and scalability in order to deal with, explore and combine a wide range of knowledge sources.

Finally, as part of the process of designing the semantic-based search and browse facilities, we carried out a small experiment with using the learnt domain concepts to discover automatically who is the most suitable GATE developer to address for a given problem. A subset of the GATE forum postings were analysed to identify all responses by GATE developers, whose names were supplied as an input to the system. The result was an association of domain concepts, developer names (as initials), and frequency of answers. Some examples are: POS tagger (DM (43 postings), IR (12)), Jena ontologies (VT (45), KB (6), IR (2)). As already discussed in Section 2, this information can help new developers to identify who they should consult when working on a given topic. Conversely, the assignment of GATE concepts to forum postings will enable our system to provide developers with the facility to be notified only of postings related to their area of expertise.

7 Future Work

The next step in the development of our system is to implement relation learning. In the first instance, we'll focus on learning the class hierarchy, i.e., *isa* relations between concepts. We will experiment with several methods:

- exploiting term compositionality [14], where if the lexical term of one concept appears as the ending of the lexical term of another concept, then the first concept is more generic than the second one. For example, `JAPE rule` is a sub-class of the concept `rule`.
- lexical patterns, such as these used in Pankow [5].
- exploiting the class inheritance hierarchy of the source code, which would for example indicate that a POS tagger is a sub-class of language analysis component.

The main outstanding challenge yet to be addressed will be in adapting the ontology learning techniques to deal with the dynamic nature of software artifacts and in implementing the corresponding semantic-based access search and browse methods.

Finally, we plan to perform a more thorough evaluation of our concept and relation learning methods. The first step will be to build a gold standard domain ontology, starting from the automatically learnt concepts and relations. Then the automatically created ontology will be evaluated against the gold standard by using metrics such as lexical and taxonomic precision and recall [8].

Acknowledgements

This work is partially supported by the EU-funded TAO project (IST-2004-026460).

References

1. A. Ankolekar, K. Sycara, J. Herbsleb, and R. Kraut. Supporting Online Problem Solving Communities with the Semantic Web. In *Proc. of WWW*, 2006.
2. K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Engineering*, 10(3/4):349—373, 2004. `http://www.gate.ac.uk/sale/jnle-sale/subs/BONTCHEVA--jnle-final.pdf`.
3. P. Buitelaar, P. Cimiano, and B. Magnini. *Ontology learning from text: Methods, applications and evaluation*. IOS Press, 2005.
4. P. Buitelaar, D. Olejnik, and M. Sintek. A Protege Plug-in for Ontology Extraction from Text Based on Linguistic Analysis. In *Proceedings of the 1st European Semantic Web Symposium*, 2004.
5. P. Cimiano, S. Handschuh, and S. Staab. Towards the Self-Annotating Web. In *Proc. of the 13th International Conference on World Wide Web (WWW'04)*, 2004.
6. H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, 2002.
7. B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht. Self-Organized Reuse of Software Engineering Knowledge Supported by Semantic Wikis. In *Workshop on Semantic Web Enabled Software Engineering (SWESE)*, Galway, Ireland, 2005.
8. K. Dellschaft and S. Staab. On How to Perform a Gold Standard Based Evaluation of Ontology Learning. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, Athens, GA, USA, to appear.
9. M. Dowman, V. Tablan, H. Cunningham, and B. Popov. Web-assisted annotation, semantic indexing and search of television and radio news. In *Proceedings of the 14th International World Wide Web Conference*, Chiba, Japan, 2005. `http://gate.ac.uk/sale/www05/web-assisted-annotation.pdf`.
10. P. Haase, Y. Sure, and D. Vrandecic. D3.1.1 Ontology Management and Evolution Survey, Methods and Prototypes. Technical report, SEKT EU Project Deliverable, 2004.
11. A. Kiryakov, B. Popov, D. Ognyanoff, D. Manov, A. Kirilov, and M. Goranov. Semantic annotation, indexing and retrieval. *Journal of Web Semantics, ISWC 2003 Special Issue*, 1(2):671–680, 2004.
12. H. Knublauch. Ramblings on Agile Methodologies and Ontology-Driven Software Development. In *Workshop on Semantic Web Enabled Software Engineering (SWESE)*, Galway, Ireland, 2005.
13. A. Maedche. *Ontology Learning fot the Semantic Web*. Kluwer Academic Publishers, Amsterdam, 2002.
14. M. Sabou. *Building Web Service Ontologies*. PhD thesis, Vrije Universiteit, 2006.
15. M. Sabou and J. Pan. Towards Improving Web Service Repositories through Semantic Web Techniques. In *Workshop on Semantic Web Enabled Software Engineering (SWESE)*, Galway, Ireland, 2005.
16. B. Sapcota. Web Service Discovery in Distributed and Heterogeneous Environments. In *International Conference on Web Intelligence (WI'05)*, Compiegne, France, 2005.
17. P. Warren, I. Thurlow, and D. Alsmeyer. Applying Semantic Technology to a Digital Library. In J. Davies, R. Studer, and P. Warren, editors, *Semantic Web Technologies*. John Wiley and Sons, 2006.