

Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach*

René Witte and Qiangqiang Li

Institut für Programmstrukturen und Datenorganisation (IPD)

Fakultät für Informatik

Universität Karlsruhe (TH), Germany

Yonggang Zhang and Juergen Rilling

Department of Computer Science and Software Engineering

Concordia University, Montréal, Canada

Abstract

Documents written in natural languages constitute a major part of the artifacts produced during the software engineering lifecycle. Especially during software maintenance or reverse engineering, semantic information conveyed in these documents can provide important knowledge for the software engineer. In this paper, we present a text mining system capable of populating a software ontology with information detected in documents. A particular novelty is the integration of results from automated source code analysis into a natural language processing (NLP) pipeline, allowing to cross-link software artifacts represented in code and natural language on a semantic level.

1 Introduction

With the ever increasing number of computers and their support for business processes, an estimated 250 billion lines of source code were being maintained in 2000, with that number rapidly increasing [28]. The relative cost of maintaining and managing the evolution of this large software base represents now more than 90% of the total cost [26] associated with a software product. One of the major challenges for software engineers while performing a maintenance task is the need to comprehend a multitude of often disconnected artifacts created originally as part of the software development process [12]. These artifacts include, among others, source code and corresponding software documents, e.g., requirements specifications, design descriptions, and user's guides. From a maintainer's perspective, it becomes essential to establish and maintain the seman-

tic connections among all these artifacts. Automated source code analysis, implemented in integrated development environments like *Eclipse*, has improved software maintenance significantly. However, integrating the often large amount of corresponding documentation requires new approaches to the analysis of natural language documents that go beyond simple full-text search or information retrieval (IR) techniques [1].

In this paper, we propose a Text Mining (TM) approach to analyse software documents at a semantic level. A particular feature of our system is its use of formal ontologies (in OWL-DL format) during both the analysis process and as an export format for the results. In combination with a source code analysis system for populating code-specific parts of the ontology, we can now represent knowledge concerning *both* code *and* documents in a single, unified representation. This common, formal representation supports further analysis of the knowledge base, like the automatic establishment of traceability links. A general overview of the proposed process is shown in Figure 1: An existing ontology of the software domain, including concepts of

*This paper is a postprint of a paper submitted to and accepted for publication in the *IET Software Journal*, Vol. 2, No. 1, 2008, and is subject to IET copyright [<http://www.iet.org>]. The copy of record is available at <http://link.aip.org/link/?SEN/2/3/1>.

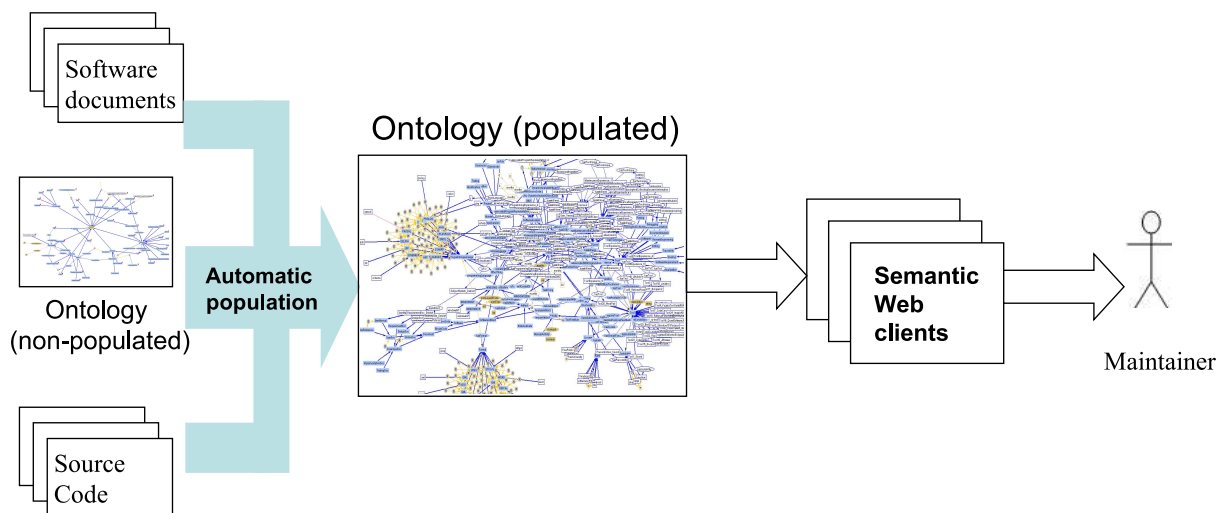


Figure 1: Ontological Text Mining of software documents for software engineering

programming languages and software documentation, is automatically populated through the analysis of existing source code and corresponding documentation. The knowledge contained in this populated ontology can then be used by a software maintainer through “Semantic Web”-enabled clients utilizing ontology queries and reasoning.

Our research presents a novel strategy to employ knowledge obtained from source code analysis for the analysis of corresponding natural language documents. The approach of combining both now allows software engineers to derive actionable knowledge from the diverse knowledge resources through ontology queries and automated reasoning, extending the applications of our research far beyond simple information extraction or retrieval.

This paper is structured as follows: In the next section, we motivate the need for integrating text mining and software engineering and describe our solution, an ontology-based program comprehension environment. Section 3 describes our text mining system for the analysis of software documents. Results from our evaluation are presented in Section 4. We then show how to apply our system for the analysis of a large open source system, uDig, in Section 5. Related work is discussed in Section 6, followed by conclusions in Section 7.

2 Ontology-Based Program Comprehension Environment

In this section, we first present a brief motivation and overview of our ontology-based software environment

and then discuss our main resource, an OWL-DL ontology, in detail.

2.1 Software Engineering and Natural Language Processing (NLP)

As software ages, the task of maintaining it becomes more complex and more expensive. Software maintenance, often also referred to as software evolution, constitutes a majority of the total cost occurring during the life span of a software system [26, 28]. Software maintenance is a difficult task complicated by several factors that create an ongoing challenge for both the research community and tool developers [10, 24]. These maintenance challenges are caused by the different representations and interrelationships that exist among software artifacts and knowledge resources [29, 30]. From a maintainer’s perspective, exploring [15] and linking these artifacts and knowledge resources becomes a key challenge [1]. What is needed is a unified representation that allows a maintainer to explore, query, and reason about these artifacts, while performing their maintenance tasks [21].

Information contained in software documents is important for a multitude of software engineering tasks (e.g., requirements engineering, software maintenance), but within this paper, we focus on a particular use case: the concept location and traceability across different software artifacts. From a maintainer’s perspective, software documentation contains valuable information of both functional and non-functional requirements, as well as information related to the application domain. This knowledge often is difficult or impossible to extract only from source code [16]. It is a well-known

fact that even in organizations and projects with mature software development processes, software artifacts created as part of these processes end up to be disconnected from each other [1]. As a result, maintainers have to spend a large amount of time on synthesizing and integrating information from various information sources in order to re-establish the traceability links among these artifacts.

Our approach is based on a common formal representation of both source code and software documentation based on an ontology in OWL-DL format [13]. Instances are populated automatically through automatic source code analysis (described in [34]) and text mining (described in this paper). The resulting, populated ontology can serve as a flexible knowledge base for further analysis. Employing *queries*, based on query languages like nRQL or SPARQL, together with automated *reasoning* provides a flexibility of code-document analysis tasks that goes far beyond current approaches working on source code or documentation in isolation.

2.2 System Architecture and Implementation Overview

In order to utilize the structural and semantic information in various software artifacts, we have developed an ontology-based program comprehension environment, which can automatically extract concept instances (e.g., classes, methods, variables) and their relations from source code and documents (Figure 2).

2.2.1 Software Ontology

An important part of our architecture is a *software ontology* that captures major concepts and relations in the software maintenance domain (right side in Figure 2). Ontologies are a commonly used technique for knowledge representation; here, we rely on the more recently introduced Web Ontology Language (OWL), which has been standardized by the World Wide Web Consortium (W3C).¹ In particular, we make use of the sub-format OWL-DL, which is based on Description Logics (DL) as the underlying formal representation. This format provides maximal expressiveness while still being complete and decidable, which is important as our environment relies on an automated *reasoner* for several knowledge manipulation tasks. For more details on DL, we refer the reader to [4].

Our software ontology consists of two sub-ontologies: a *source code* and *document* ontology,

which represent information extracted from source code and documents, respectively.

2.2.2 Semantic Web Infrastructure

As a standardized Web format, OWL ontologies are supported by a large number of (open source) tools and libraries, which allows us to create complex knowledge-based systems based on existing Semantic Web [3] infrastructure.

The software ontology was created using the OWL extension of Protégé,² a free ontology editor. Racer [9], an ontology inference engine, is integrated to provide reasoning services. Racer is a highly optimized DL system that supports reasoning about instances, which is particularly useful for the software maintenance domain, where a large amount of instances needs to be handled efficiently. Other services provided by Racer include terminology inferences (e.g., concept consistency, subsumption, classification, and ontology consistency) and instance reasoning (e.g., instance checking, instance retrieval).

2.2.3 Ontology Population Subsystems

One of the major challenges in software analysis is the large amount of information that has to be explored and analyzed as part of typical maintenance activities. Manually adding information about instances (e.g., concrete variables or methods in source code, sentences and concepts in documentation) is not a feasible solution. Thus, a prerequisite for our ontology-based approach is the development of automatic ontology population systems, which create instances for the concepts modeled in the ontology based on an analysis of existing source code and its documentation.

The automatic ontology population (Figure 2, middle) is handled by two subsystems: The source code analysis system and the text mining system. The source code ontology population subsystem is based on JDT,³ which is a Java parser provided by Eclipse. JDT reads the source code and performs common tokenization and syntax analysis to produce an Abstract Syntax Tree (AST). Our population subsystem traverses the AST created by the JDT compiler to identify concept instances and their relations, which are then passed to an OWL generator for ontology population. More details on this subsystem are available in [34]; the text mining subsystem is described in Section 3 below.

¹OWL Web Ontology Language Guide, <http://www.w3.org/TR/owl-guide/>

²Protégé ontology editor, <http://protege.stanford.edu/>

³Eclipse Java Development Tools (JDT), <http://www.eclipse.org/jdt/>

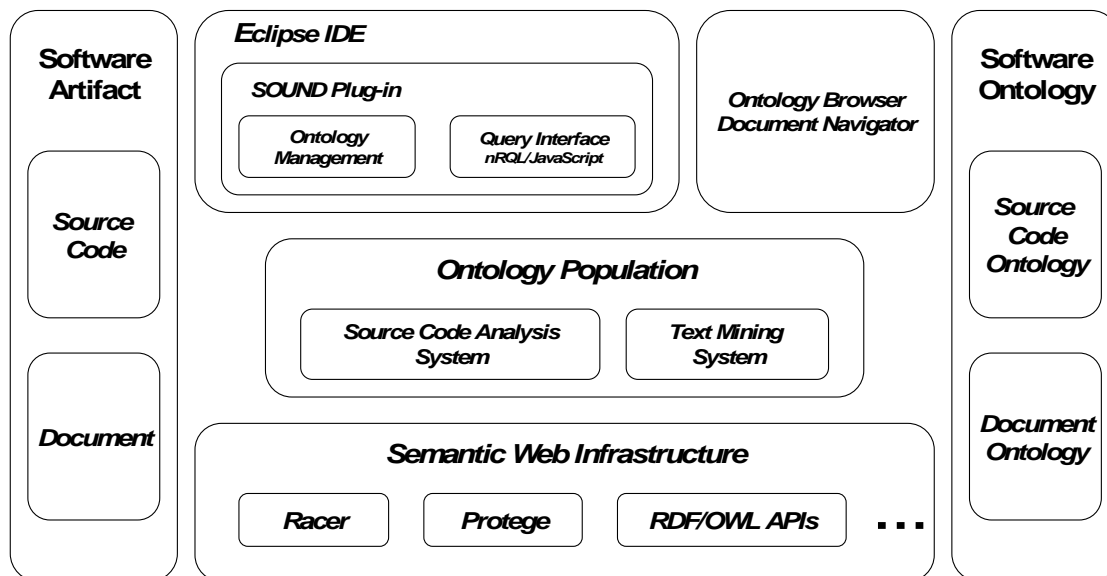


Figure 2: Ontology-based program comprehension environment overview

2.2.4 Client Integration and Querying

End users, in this case software engineers, need access to the knowledge contained in the populated ontology. We believe this access should be delivered within the tools commonly used to develop software, providing context-sensitive guidance directly relevant to the task at hand. Towards this goal, we developed a query interface for our system in form of a plug-in that provides OWL integration for Eclipse, a widely used software development platform. Through this query interface, the expressive query language nRQL provided by Racer can be used to query and reason over the populated ontology. Additionally, we integrated a scripting language, which provides a set of built-in functions and classes using the JavaScript interpreter Rhino.⁴ This language simplifies querying the ontology for software engineers not familiar with DL-based formalisms. We show examples on ontology queries in Section 5 below.

2.3 Software Document Ontology

The documentation ontology (see Figure 3 for an excerpt) consists of a large body of concepts that are expected to be discovered in software documents. These concepts are based on various programming domains, including programming languages, algorithms, data structures, and design decisions such as design patterns and software architectures. Additionally, the software documentation sub-ontology has been specifically designed for automatic population through a text mining system by adapting the ontology design requirements

⁴Rhino JavaScript interpreter, <http://www.mozilla.org/rhino/>

discussed in [33] for the software engineering domain. In particular, we included the following parts:

Text Model: represents the structure of documents, i.e., it contains concepts for sentences, paragraphs, and text positions, as well as NLP-related concepts that are discovered during the analysis process, like noun phrases (NPs) and coreference chains. These are required for anchoring detected entities (populated instances) in their originating documents. Noun phrases form the grammatical basis for finding named entities, which add the *semantics* to the detected information by labeling NPs with ontological concepts, like “class,” “method,” or “variable.”

Coreference chains connect all entities in a document that are semantically equivalent. For example, an entity, like a single method “*suite()*” can appear multiple times within a document, with different textual representations (e.g., “*the suite() method*,” “*this method*,” “*it*”). A coreference chain connects these lexically different, but semantically equivalent occurrences.

Lexical Information: facilitates the detection of entities in documents. Examples are names of common design patterns (“Bridge,” “Adapter,” etc.), programming language-specific keywords (“int,” “extends,” etc.), and architectural styles (“layered architecture,” “client/server,” etc.).

Lexical Normalization Rules: these rules transform entity names as they appear in a document to a canonical name that can be used for ontology pop-

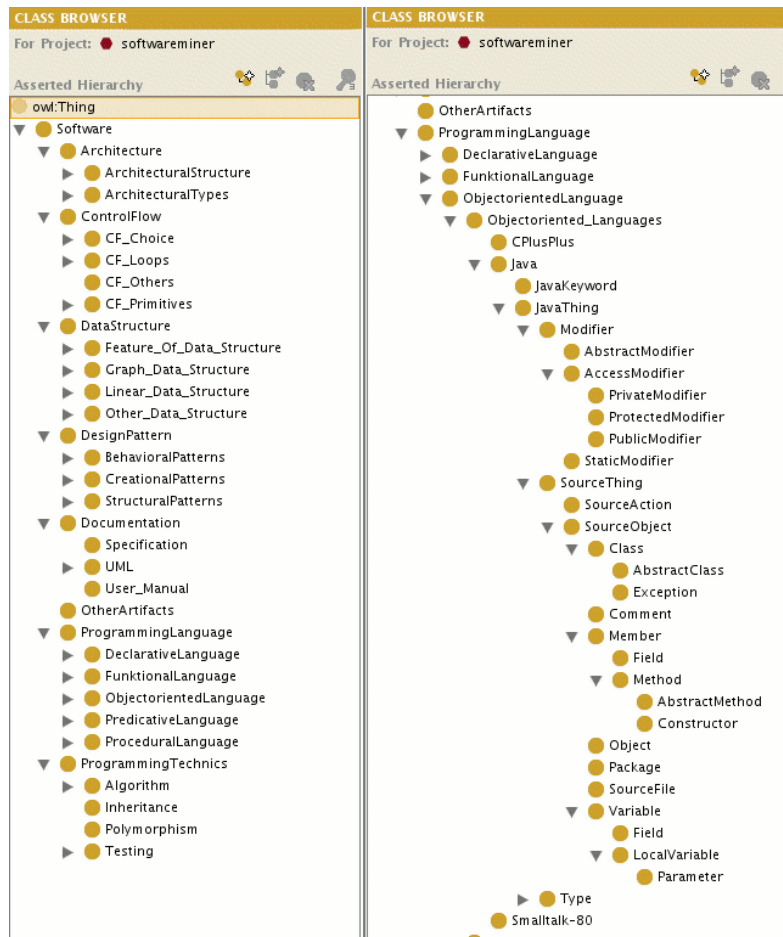


Figure 3: Excerpt of the software documentation sub-ontology used for software document analysis and NLP result export

ulation. For example, the same method name can be referenced by a multitude of textual representations, like “*the suite() method*” or “*method suite()*.” Finding the canonical name “*suite()*” from these different representations is important for further automated analyses, e.g., document/source code traceability. This is achieved through normalization rules, which are specific for a certain ontology class.

Relations: between the classes, including the ones modeled in the source code ontology. Ontology relations allow us to automatically restrict NLP-detected relations to semantically valid ones. For example, a relation like `<variable> implements <interface>`, which can result from parsing a grammatically ambiguous sentence, can be filtered out since it is not supported by the ontology.

Source Code Entities: that have been automatically populated through source code analysis can also

be utilized for detecting corresponding entities in documents, as we describe below.

How these different pieces of knowledge contribute to a detailed semantic analysis of software documents is described in the next section.

3 Ontology Population through Text Mining

We developed our text mining system for populating the software documentation ontology based on the GATE⁵ (*General Architecture for Text Engineering*) framework [7]. Our system is component-based, utilizing both standard tools shipped with GATE and custom components developed specifically for software text mining. An overview of the workflow is shown in Figure 4.

⁵GATE, <http://gate.ac.uk/>

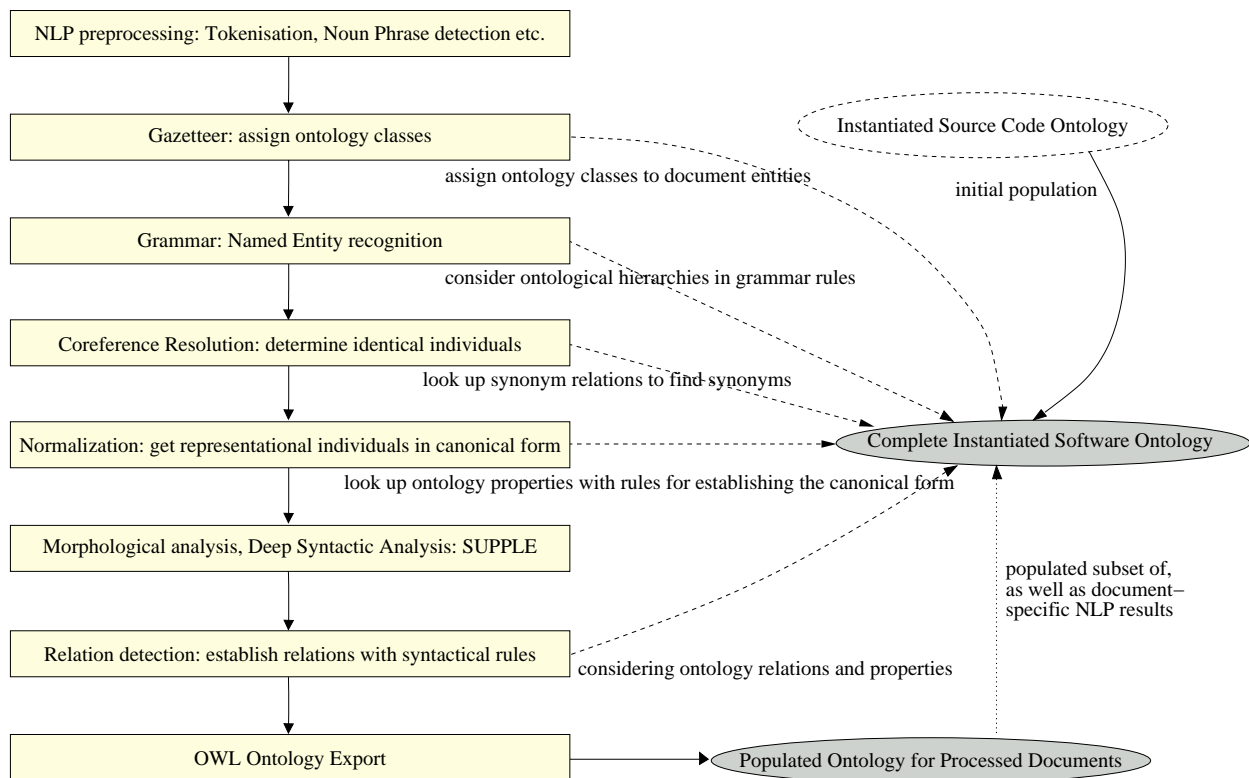


Figure 4: Workflow of the software text mining subsystem

3.1 Preprocessing

The processing pipeline starts with a number of standard preprocessing steps (Figure 4, top left). These generate annotations and data structures required for the more involved natural language analysis tasks performed later on.

The first step is *Tokenization*. Similarly to programming languages, tokenization splits the input text into individual tokens, separated by whitespace or special characters. *Sentence Splitting* then detects sentence boundaries between tokens using a set of rules, e.g., to avoid false splits at abbreviations or other occurrences of a full stop not indicating an end-of-sentence. *Part-of-Speech (POS) Tagging* works on a sentence basis and adds a POS tag to each token, identifying its class (e.g., noun, verb, adjective). Here, we use the Hepple tagger included in the GATE distribution. Based on the POS tags, *Chunking* modules analyze the text for Noun Phrase (NP) and Verb Group (VG) chunks. For detecting NPs, we use the open source MuNPEX chunker⁶ and for detecting VGs, we rely on the verb grouper module that comes with GATE.

For more details on these steps, we refer the reader

⁶Multi-lingual Noun Phrase Extractor (MuNPEX), <http://www.ipd.uni-karlsruhe.de/~durm/tm/munpex/>

to the GATE user's guide.⁷

3.2 Ontology Initialization

While analysing documents specific to a source code base, our text mining system can take instances detected by the automatic code analysis into account. This is achieved in two steps: first, the source code ontology is populated with information detected through static and dynamic code analysis [34]. This step adds instances like method names, class names, or detected design patterns to the software ontology. In a second step, we use this information as additional input to the *OntoGazetteer* component for named entity recognition.

3.3 Named Entity Detection

The basic process in GATE for recognizing entities of a particular domain starts with the *gazetteer* component. It matches given lists of terms against the tokens of an analysed text and, in case of a match, adds an annotation named `Lookup` whose features depend on the list where the match was found. Its ontology-aware counterpart is the *OntoGazetteer*, which incorporates mappings between its term lists and ontology classes and assigns the proper class in case of a term match. For

⁷GATE user's guide, <http://gate.ac.uk/sale/tao/index.html>

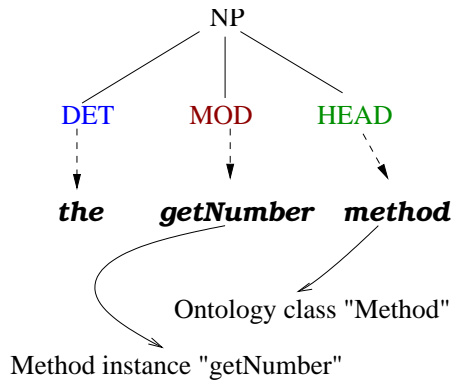


Figure 5: Combining ontology and noun phrase chunks for named entity detection

example, using the instantiated software ontology, the gazetteer will annotate the text segment *method* with a `Lookup` annotation that has its `class` feature set to “Method.” Here, incorporating the results from automatic code analysis can significantly boost recall (cf. Section 4), since entity names in the software domain typically do not follow naming rules.⁸

In a second step, grammar rules written in the JAPE⁹ language are used to detect and annotate complex named entities. Those rules can refer to the `Lookup` annotation generated by the `OntoGazetteer`, and also evaluate the ontology directly. For example, when performing a comparison like `class=="Keyword"` in a grammar rule, the complete concept hierarchy in the ontology is considered in this comparison and as a result also includes a match for a Java keyword, since a Java keyword is a subclass of the concept “Keyword” in the ontology. This feature significantly reduces the overhead for grammar development and testing.

The developed JAPE rules combine ontology-based lookup information with noun phrase (NP) chunks to detect semantic units. NP chunking is performed using the `MuNPEx` chunker,¹⁰ which relies mostly on part-of-speech (POS) tags, but can also take the lookup information into account. This way, it can prevent bad NP chunks caused by mis-tagged software entities (e.g., method names or program keywords tagged as verbs). Essentially, we combine two complemen-

tary approaches for entity detection: A *keyword*-based approach, relying on lexical information stored in the documentation ontology (see above). For example, the text segment “*the getNumber() method...*” will be annotated with a lookup information indicating that the word *method* belongs to the ontology class “Method.” Likewise, the same segment will be annotated as a single noun phrase, showing determiner (“*the*”), modifier (“*getNumber()*”), and head noun (“*method*”). Using an ontology-based grammar rule implemented in JAPE, we can now combine these two sources of information and semantically mark the NP as a method (Figure 5). Similar rules are used to detect variables, class names, design patterns, or architectural descriptions. Note that this approach does not need to know about “*getNumber()*” being a method name; this fact is derived from a combination of grammatical (NP chunks) and lexical (ontology) information.

The second approach relies on source code analysis results stored in the initialized software ontology (see Section 3.7). Every method, class, package, etc. name will be automatically represented by an instance in the source code sub-ontology and can thus be used by the `OntoGazetteer` for entity detection. This applies also in case when these instances appear outside a grammatical construct recognized by our hand-crafted rules. This is especially useful for analysing software documents in conjunction with their source code, the primary scenario our system was designed for.

3.4 Coreference Resolution

We use a fuzzy set theory-based coreference resolution system [31] for grouping detected entities into *coreference chains*. Each chain represents an equivalence class of textual descriptors occurring within or across documents. Not surprisingly, our fuzzy heuristics developed originally for the news domain (e.g., using *WordNet*) were particularly ineffective for detecting coreference in the software domain. Hence, we developed an extended set of heuristics dealing with both pronominal and nominal coreferences.

For *nominal coreferences* (i.e., references between full noun phrases, excluding pronouns), we rely on three main heuristics. The first is based on simple string equality (ignoring case). The second heuristic establishes coreference between two entities if they become identical when their NPs’ HEAD and MOD slots are inverted, as in “*the selectState() method*” and “*method selectState()*”. The third heuristic deals with a number of grammatical constructs often used in software documents that indicate synonymous entities. For example,

⁸In many other domains, like in biology, an entity’s type can be (partially) derived from its lexical form, e.g., every noun ending with the letters *-ase* must be an *enzyme*. But no such rules exist in the software domain for, e.g., method or variable names.

⁹JAPE is a regular-expression based language for writing grammars over annotation graphs, from which finite-state transducers are generated by a GATE component.

¹⁰Multi-Lingual Noun Phrase Extractor (MuNPEx), <http://www.ipd.uka.de/~durm/tm/munpex/>

Table 1: Lexical normalization rules for various ontology classes

Ontology Class	H	DH	MH(cM)	MH(cH)	DMH(cM)	DMH(cH)
Class	H	H	H	lastM	H	lastM
Method	H	H	H	lastM	H	lastM
Package	H	H	H	lastM	H	lastM
OO_Object	H	H	H	lastM	H	lastM
LayeredArchitecture	H	H	MH	MH	MH	MH
Layer	H	H	MH	MH	MH	MH
AbstractFactory	H	H	MH	MH	MH	MH
OO_Interface	H	H	H	lastM	H	lastM

in the text fragment “. . . we have an action class called *ViewContentAction*, which is invoked.” we can identify the NPs “an action class” and “*ViewContentAction*” as being part of the same coreference chain. This heuristic only considers entities of the same ontology class, connected by a number of pre-defined relation words (e.g., “named”, “called”), which are also stored in the ontology.

For *pronominal resolution* (i.e., references including a pronoun), we implemented a number of simple sub-heuristics dealing only with 3rd person singular and plural pronouns: *it*, *they*, *this*, *them*, and *that*. The last three can also appear in qualified form (*this method*, *that constructor*). We employ a simple resolution algorithm, searching for the closest anaphorical referent that matches the case and, if applicable, the semantic class.

3.5 Normalization

Normalization needs to decide on a canonical name for each entity, like a class or method name. This is important for ontology population, as an instance, like of the ontology class “Method,” should reflect only the method name, omitting any additional grammatical constructs like determiners or possessives. Thus, a named entity like “*the static TestCase() class*” has to be normalized to “*TestCase*” before it can become an instance (ABox) of the concept *Class* (TBox) in the populated ontology.

This step is performed through a set of lexical normalization rules, which are stored with their corresponding classes in the software document sub-ontology, allowing us to inherit rules through subsumption. Table 1 shows a number of these rules for various ontology classes: D, M, H refer to determiner, modifier, and head, respectively, and $c(x)$ denotes the ontology class of a particular slot; the table entry determines what part of a noun phrase is selected as the normalized form, which is then stored as a feature

`instanceName` in the entity’s annotation, as shown in Figure 6.

3.6 Relation Detection

The next major step is the detection of *relations* between entities, e.g., to find out which interface a class is implementing, or which method belongs to which class. Relation detection in our system is again achieved by combining two complementary approaches, as shown in Figure 7: a set of hand-crafted grammar rules implemented in JAPE and a deep syntactic analysis using the SUPPLE parser. Afterwards, detected relations are filtered through the software ontology to erase semantically invalid results. We now describe these steps in detail.

3.6.1 Rule-Based Relation Detection

Similarly to entity recognition, rule-based relation detection is performed in a two-step process, starting with the verb groups (VGs) detected in the preprocessing step discussed above. In addition, our ontology contains lexical information for the modeled relations (cf. Section 2.3), e.g., “implements” for the class-interface relation or “provides” for the class-method relation. This allows to detect candidate relation words in a text using the OntoGazetteer component. The intersection of both—VGs containing a detected relation word—are candidates for further processing. For example, given a sentence fragment like “*the Test class provides a static suite() method*,” the token “provides” would be marked as both a VG (containing only a single, active verb) and a relation candidate (for the class-method relation).

In a second step, hand-crafted grammatical rules for relation detection are run over the relation candidates to find, e.g., relations between classes and design patterns or classes and methods. These rules are also implemented in JAPE and make use of the subsumption hierarchy in the ontology. For ex-

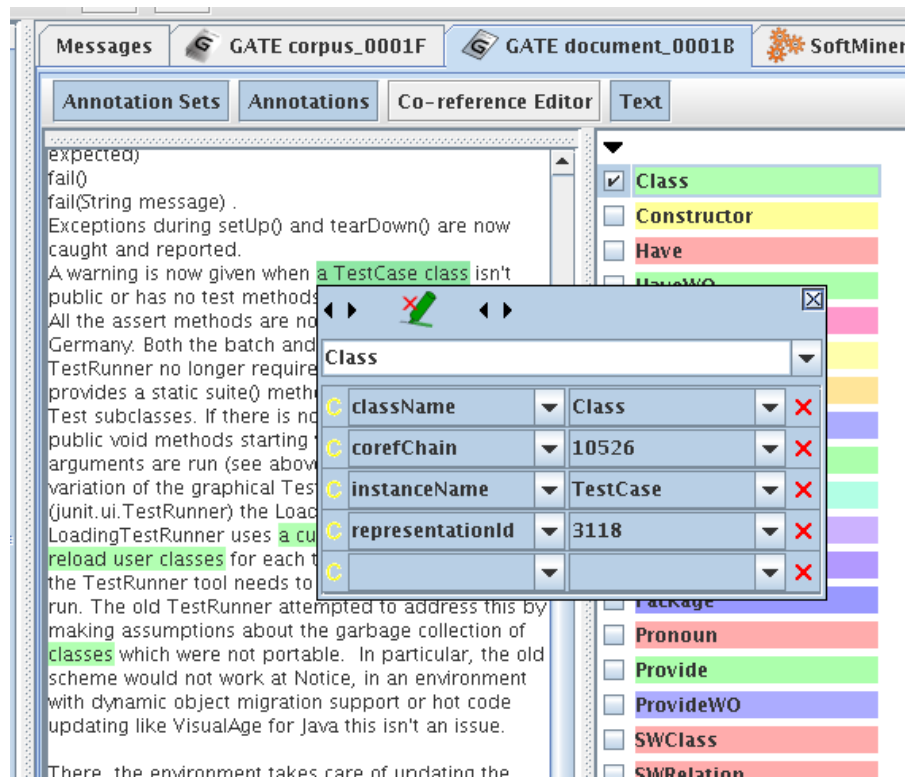


Figure 6: Java *class* entity detected through NLP displayed in GATE showing the normalized instanceName for ontology export

ample, the above sentence would be matched by the simple rule $\langle \text{Software-Entity} \rangle \text{ relation } \langle \text{Software-Entity} \rangle$. Using the voice information (active/passive) provided by the VG chunker, we can then assign subject/object slots for the entities participating in a relation. The final result of this step for the above example is the relation *provides* (Class:Test, Method:static suite()).

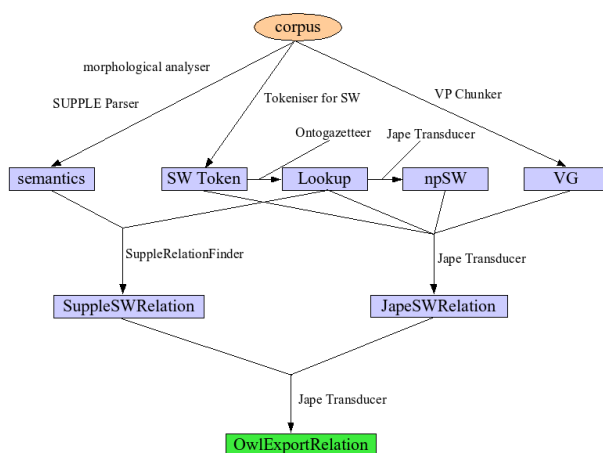


Figure 7: Finding relations between software entities

3.6.2 Deep Syntactic Analysis

For a deep syntactic analysis, we currently employ the SUPPLE parser [8], which is integrated into GATE through a wrapper component. SUPPLE is a general-purpose bottom-up chart parser for feature-based context-free phrase structure grammars, implemented in Prolog. It produces syntactic as well as semantic annotations for a given sentence. Grammars are applied sequentially. After each layer of grammatical analysis, only the best parse is selected for the next step. Thus, SUPPLE avoids the multiplication of ambiguities throughout the grammatical and semantical analysis, which comes with the trade-off of losing alternatives that could be resolved in later analysis steps. The identification of verbal arguments and attachment of nominal and verbal post-modifiers, such as prepositional phrases and relative clauses, is done conservatively. Instead of producing all possible analyses or using probabilities to generate the most likely analysis, SUPPLE only offers a single analysis that spans the input sentence only if it can be relied on to be correct, so that in many cases only partial analyses are produced. This strategy generally has the advantage of higher precision, but at the expense of lower recall. SUPPLE outputs a logical form, which is then matched with the entities detected

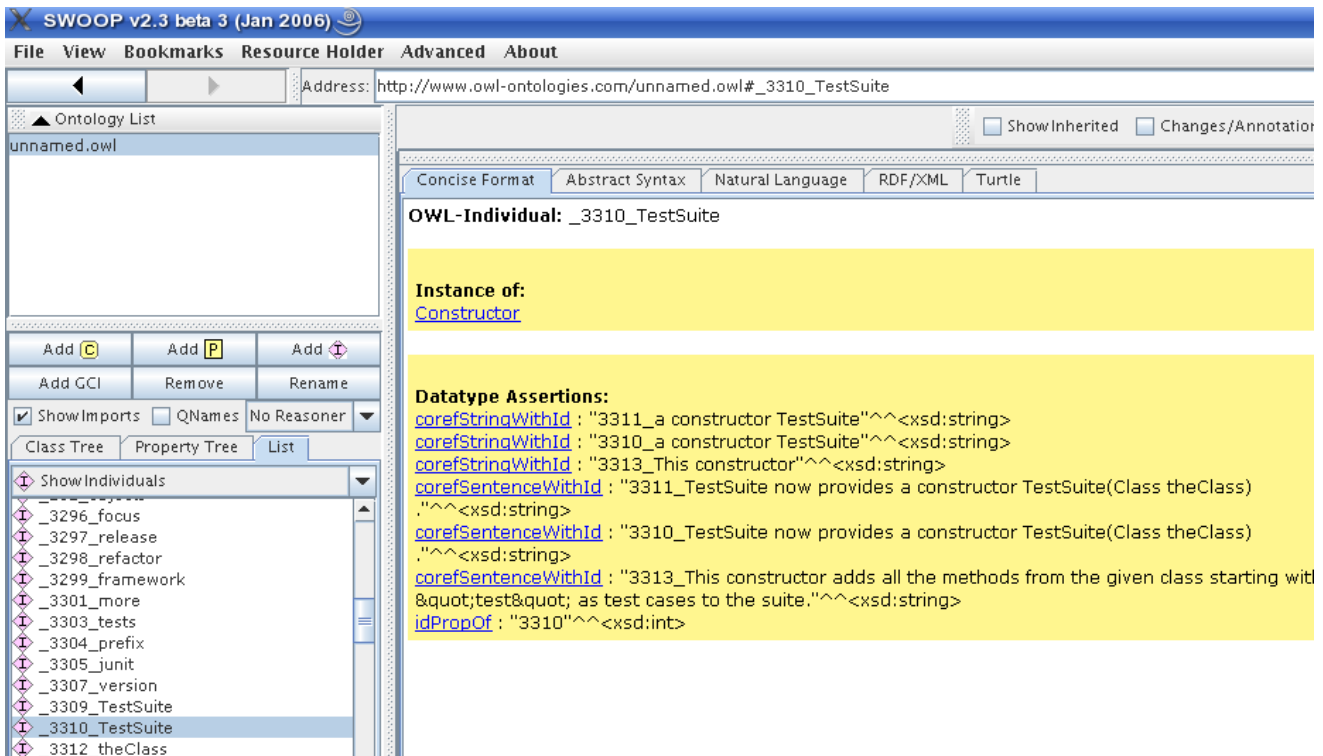


Figure 10: Text Mining results by ontology population: a detected Java *Constructor* instance browsed in SWOOP showing information about coreferences and corresponding sentences

3.7 Ontology Export

Finally, the instances found in the document and the relations between them are exported to an OWL-DL ontology. Note that entities provided by applying the source code analysis step are only exported in the document ontology if they have also been detected in a text (cf. Figure 4).

In our implementation, ontology population is done by a custom GATE component, the *OwlExporter*, which is application domain-independent. It collects two special annotations, *OwlExportClass* and *OwlExportRelation*, which specify instances of classes and relations (i.e., object properties), respectively. These must in turn be created by application-specific components, since the decisions as to which annotations have to be exported, and what their OWL property values are, depend on the domain.

The class annotation carries the name of the class, a name for the instance (the normalized name created previously), and the GATE internal ID of an annotation representing the instance in the document. If there are several occurrences of the same entity in the document, the final representation annotation is chosen from the ones in the coreference chain by the component creating the *OwlExportClass* annotation. In case of the

software text mining system, a single representative has to be chosen from each coreference chain. Remember that one chain corresponds to a single semantic unit, so the final, exported ontology must only contain one entry for, e.g., a method, not one instance for every occurrence of that method in a document set. We select the representative using a number of heuristics, basically assuming that the longest NP that has more slots (DET, MOD, HEAD) filled is also the most salient one.

From this representative annotation, all further information is gathered. After reading the class name, the *OwlExporter* queries the ontology via the Jena¹¹ framework for the class properties and then searches for equally named features in the representation annotation, using their values to set the OWL properties.

An example for an exported instance (an OWL individual) can be seen in Figure 10. Note the detailed semantic information obtained about the textual reference “*TestSuite*.” Besides the semantic type (ontology class *Constructor*), the text mining system recorded the different occurrences of this entity throughout a document. In particular, the last sentence “*This constructor adds all the methods...*” contains knowledge important for a software maintainer that would not have been de-

¹¹Jena Semantic Web Framework for Java, <http://jena.sf.net/>

ected by simple full-text search or IR methods, as it does not contain the string “TestSuite” itself. This illustrates the importance of advanced NLP for software documents, in this case, coreference resolution.

4 Evaluation

In this section, we present an evaluation of our approach. We first introduce our analysis strategy and metrics for readers that are unfamiliar with the evaluation of text mining systems in Sections 4.1 and 4.2. The different parts of the system—for entity recognition, entity normalization, and relation detection—are then evaluated in detail in Sections 4.3–4.5.

4.1 Evaluation Strategy and Corpus

Generally, the effectiveness of a text mining system is measured by running it against a corpus (set of documents), where the expected results are known. This allows to compare the output of a system (the *reply*) with the correct, expected results (the *model*). Of course, since no system is available to produce 100% correct results, the expected reply has to be provided by manually *annotating* the documents from the corpus. This so-called *gold standard* then serves as the reference for automated evaluation metrics, allowing a comparison of different approaches on the same data and ensuring repeatability. The downside is that manual annotation is time-consuming and thus expensive, and can also suffer from errors introduced by annotators.

So far, we evaluated our text mining subsystem on two collections of texts: a set of 5 documents (7743 words) from the Java 1.5 documentation for the *Collections* framework¹² and a set of 7 documents (3656 words) from the documentation of the uDig¹³ geographic information system (GIS). The document sets were chosen because of the availability of the corresponding source code. Both sets were manually annotated for named entities, including their ontology classes and normalized form, as well as relations between the entities.

4.2 Evaluation Metrics

The metrics *precision*, *recall* and *F-measure* are commonly used in the evaluation of NLP systems. They have been adapted from their Information Retrieval (IR) [5] counterparts. Precision and recall are based on

¹²Java Collections Framework Documentation, <http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>

¹³uDig GIS Documentation, <http://udig.refractory.net/>

the notion of entities that have been correctly found by a system (*correct*), entities that are *missing*, and entities wrongly detected by a system, i.e., *spurious* entities or false positives. To capture partially correct results, i.e., entities where gold standard and system response overlap without being coextensive (matching 100%), a third category *partial* is introduced in addition to correct and spurious results.

Following the definition implemented by the GATE annotation evaluation tool,¹⁴ we can define *precision* as the ratio of correctly detected entities over all retrieved entities:

$$\text{Precision} = \frac{\text{Correct} + \frac{1}{2}\text{Partial}}{\text{Correct} + \text{Spurious} + \frac{1}{2}\text{Partial}} \quad (1)$$

Sometimes the *error rate* is used instead of precision, which is simply defined as $1/\text{Precision}$. Similarly, *recall* is the ratio of correctly detected entities over all correct entities:

$$\text{Recall} = \frac{\text{Correct} + \frac{1}{2}\text{Partial}}{\text{Correct} + \text{Missing} + \frac{1}{2}\text{Partial}} \quad (2)$$

The F-measure combines both precision and recall using their harmonic mean:

$$\text{F-Measure} = \frac{(\beta^2 + 1)\text{Precision} \cdot \text{Recall}}{(\beta^2\text{Recall}) + \text{Precision}} \quad (3)$$

Here, β is an adjustable weight to favour precision over recall (with $\beta = 1$, both are weighted equally).

For more details on the evaluation of text mining and NLP, we refer to the evaluation section in the GATE documentation, “*Performance Evaluation of Language Analysers*,” as well as Section 8.1 in [17].

4.3 Named Entity Recognition Evaluation

For NE detection, we computed the precision, recall, and F-measure results using the output of our system with the gold standard. A named entity was only counted as correct if it matched both the textual description and ontology class. Table 2 shows the results for two experiments: first running only the text mining system over the corpora (left side) and second, performing the same evaluation after running the code analysis, using the populated source code ontology as an additional resource for NE detection as described above. As can be seen, the text mining system achieves a very high precision (90%) in the NE detection task, with a recall of 62%. With the imported source code instances,

¹⁴GATE documentation, <http://gate.ac.uk/documentation.html>

Table 2: Evaluation results: Entity recognition and normalization performance

Corpus	Text Mining Only				With Source Code Ontology			
	<i>P</i>	<i>R</i>	<i>F</i>	<i>A</i>	<i>P</i>	<i>R</i>	<i>F</i>	<i>A</i>
Java Collections	0.89	0.67	0.69	75%	0.76	0.87	0.79	88%
uDig	0.91	0.57	0.59	82%	0.58	0.87	0.60	84%
Total	0.90	0.62	0.64	77%	0.67	0.87	0.70	87%

Table 3: Evaluation results: Relation detection performance

Corpus	Before Filtering			After Filtering			
	<i>P</i>	<i>R</i>	<i>F</i>	<i>P</i>	<i>R</i>	<i>F</i>	ΔP
Text Mining Only							
Java Collections	0.35	0.24	0.29	0.50	0.24	0.32	30%
uDig	0.46	0.34	0.39	0.55	0.34	0.42	16%
Total	0.41	0.29	0.34	0.53	0.29	0.37	23%
With Source Code Ontology							
Java Collections	0.14	0.36	0.20	0.20	0.36	0.25	30%
uDig	0.11	0.41	0.17	0.24	0.41	0.30	54%
Total	0.13	0.39	0.19	0.22	0.39	0.23	41%

these numbers become reversed: the system can now correctly detect 87% of all entities, but with a lower precision of 67%.

The drop in precision after code analysis is mainly due to two reasons. Since names in the software domain do not have to follow any naming conventions, simple nouns or verbs often used in a text will be mis-tagged after being identified as an entity appearing in a source code. For example, the Java method *sort* from the collections interface will cause all instances of the word “sort” in a text to be marked as a method name. Another precision hit is due to the current handling of class constructor methods, which are typically identical to the class name. Currently, the system cannot distinguish the class name from the constructor name, assigning both ontology classes (i.e., *Constructor* and *OO_Class*) for a text segment, where one will always be counted as a false positive.

Both cases require additional disambiguation strategies when importing entities from source code analysis, which are currently under development. However, the current results already underline the feasibility of our approach of integrating code analysis and NLP.

4.4 Entity Normalization Evaluation

We also evaluated the performance of our lexical normalization rules for entity normalization, since correctly normalized names are a prerequisite for the correct population of the result ontology. For each entity,

we manually annotated the normalized form and computed the accuracy *A* as the percentage of correctly normalized entities over all correctly identified entities. Table 2 shows the results for both the system running in text mining mode alone and with additional source code analysis. As can be seen from the table (columns *A*), the normalization component performs rather well with 77% and 87% accuracy for the analysis with/without source code ontology import, respectively.

4.5 Relation Detection Evaluation

Not surprisingly, relation detection was the hardest sub-task within the system. Like for entity detection, we performed two different experiments, with and without source code analysis results. Additionally, we evaluate the influence of the semantic relation filtering step using our ontology as described above. The results are summarized in Table 3. As can be seen, the current combination of rules with the SUPPLE parser achieves only average performance. However, the increase in precision (ΔP) when applying the filtering step using our ontology is significant: upto 54% better than without semantic filtering.

The errors leading to missing and spurious relations are mainly due to the unchanged SUPPLE parser rules, which have not yet been adapted to the software domain. Also, the conservative approach of the SUPPLE parser when linking grammatical constituents, especially the attachment of prepositional phrases, leads to further

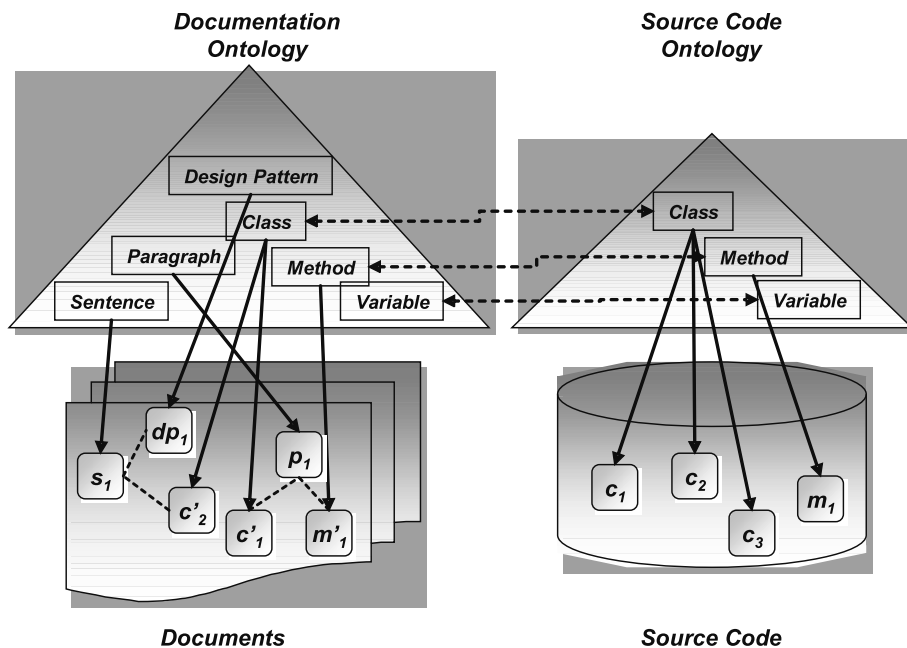


Figure 11: Linking the automatically populated source code and software documentation ontologies

missing predicate-argument structures. We are currently experimenting with different parsers (RASP and MiniPar) and are also adapting the SUPPLE grammar rules in order to improve the detection of predicate-argument structures.

5 Applications

In this section, we show how a software engineer can apply the developed system for program comprehension tasks [34]. In particular, we focus on a specific use case, the recovery of traceability links [23].

5.1 Linking Software and Documentation Ontology

Having both source code and documents represented in the form of an ontology allows us to link instances from source code and documentation using existing approaches from the field of ontology alignment [22]. Ontology alignment techniques try to match ontological information from different sources on conceptual or/and instance levels. Since our documentation ontology and source code ontology share many concepts from the programming language domain, such as *Class* or *Method*, the problem of conceptual alignment has been minimized. This research therefore focuses more on matching instances that have been discovered both from source analysis and text mining (Figure 11).

As described above, our text mining system can additionally take the results of the source code analysis as input when detecting named entities. This allows us directly connect instances from the source code and document ontologies. For example, our source code analysis tool may identify c_1 and c_2 as classes, and this information is used by the text mining system to identify the named entities c'_1, c'_2 and their associated information in the documents. As a result, source code entities c_1 and c_2 are now linked to their occurrences in the documents (c'_1 and c'_2), as well as other information about the two entities mentioned in the document, such as design patterns, architectures, etc.

5.2 Querying the Combined Ontology

After source code and documentation ontology are linked, users can perform ontological queries on both documents and source code regarding properties of the classes c_1 and c_2 . For example, a user can execute a query to retrieve document passages that describe both c_1 and c_2 or design pattern descriptions referring to the class that contains the class currently analyzed. Note that the alignment process might also identify inconsistencies—the documentation might list a method for a different class, for example—which are detected through the alignment process and registered for further review by the user. In addition, users can always manually define new concepts/instances and relations in both ontologies to establish the links that

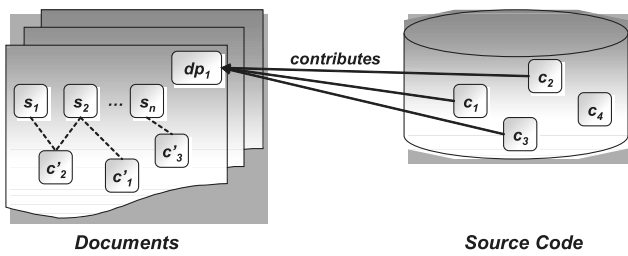


Figure 12: Manually adding relations to the result ontologies

cannot be detected by the automated alignment. For example, as Figure 12 shows, the text mining system may detect an instance of *DesignPattern*, dp_1 . A user can now manually create the relations between the pattern and classes that contribute to it (e.g., c_1 , c_2 , and c_3) through our query interface. The newly created links then become an integrated part of the ontology, and can be used to, for example, retrieve all documents related to the pattern (i.e., s_1 , s_2 , ..., s_n).

Furthermore, documents can not only be linked to source code, but also to design-level concepts that relate to particular reverse engineering tasks. For example, in contrast to the serialized view of software documents, i.e., sentence by sentence, or paragraph by paragraph, the formal ontological representation of software documentation also provides the ability to create hierarchical document views (or *slices*), comparable to context-sensitive automatic summarization [32]. Using the classification service of the ontology reasoner, one can classify document pieces that relate to a specific concept or a set of concepts (Figure 13). For example, the Visitor Pattern documents can be considered as all text paragraphs that describe/contain information related to the concept “Visitor pattern.” The newly established concept *VisitorPatternDoc* can then be used to retrieve paragraphs that relate to the visitor pattern:

$$\text{VisitorPatternDoc} \equiv \text{Paragraph} \sqcap \exists \text{contains.Visitor}$$

Similarly, a new concept *HighLevelDoc* can also be defined to retrieve all documents that contain high-level design concepts, such as *Architecture* or *DesignPattern*. The ontology reasoner can automatically classify documents according to the concept definition:

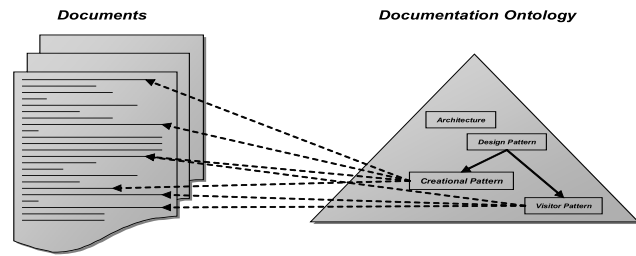
$$\begin{aligned} \text{HighLevelDoc} &\equiv \text{DocumentFile} \\ &\sqcap \exists \text{contains.}(\text{Architecture} \sqcup \text{DesignPattern}) \end{aligned}$$


Figure 13: Recombining information from different documents describing a given software entity

5.3 Case Study

We have been performing additional experiments on the large open source Geographic Information System (GIS), uDig. The uDig system is implemented as a set of plug-ins that provides geographic information management integration on top of the Eclipse platform. Links between the uDig implementation and its documentation (see Section 4) are recovered by first performing source code analysis to populate the source code ontology. The resulting ontology contains instances of *Class*, *Method*, *Field*, etc., and their relations, such as inheritance and invocation, which are then used to initialize the documentation ontology as described in Section 3.2. Through the text mining subsystem, a large number of Java language concept instances (individuals) are discovered in the documents, as well as design-level individuals, such as design patterns or architectural styles [27]. The ontology alignment rules are then applied to link both the documentation ontology and the source code ontology. Part of our initial result is shown in Figure 14; the corresponding sentences are:¹⁵

Sentence_2544: “For example if the class *FeatureStore* is the target class and the object that is clicked on is a *IGeoResource* that can resolve to a *FeatureStore* then a *FeatureStore* instance is passed to the operation, not the *IGeoResource*.”

Sentence_712: “Use the visitor pattern to traverse the AST.”

Figure 14 shows that in the uDig documents, our text mining system was able to discover that a sentence (sentence_2544) contains both the *class* instance `_4098.FeatureStore` and `_4100.IGeoResource`. Both of these instances can be linked to the instances in source code ontology, `org.geotools.data.FeatureStore` and

¹⁵Numbers in instance names are internal IDs generated by the ontology population process (see Section 3.7).

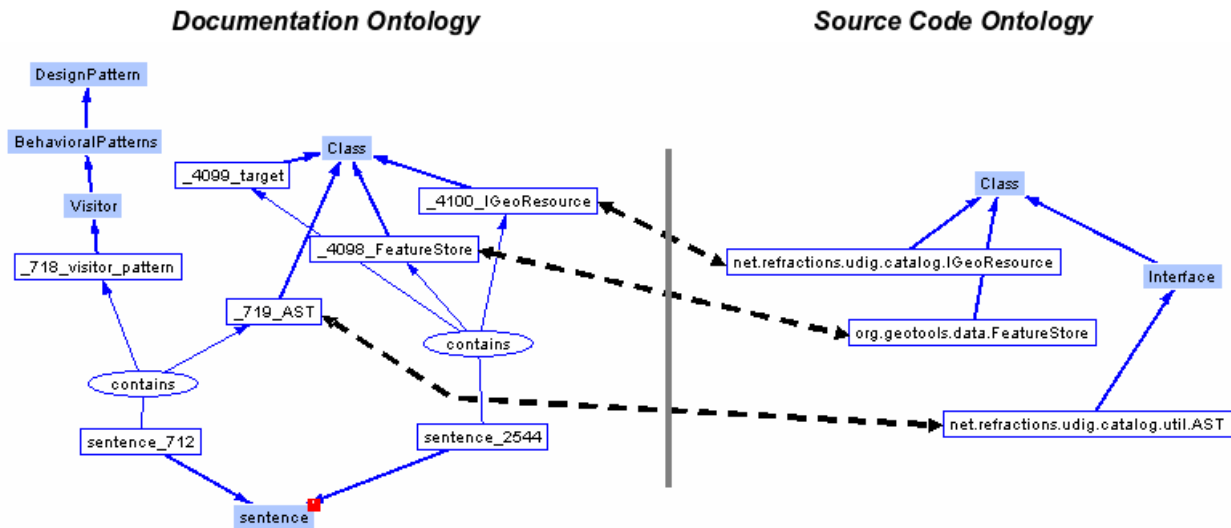


Figure 14: Automatic recovery of traceability links between source code and documentation

`net.refractions.udig.catalog.IGeoResource`, respectively. In addition, in another sentence (sentence_712), a class instance (`_719_AST`) and a design pattern instance (`_718_visitor_pattern`) are also identified. Similarly, instance `_719_AST` can then be linked to the `net.refractions.udig.catalog.util.AST` interface in the source code ontology.

After the source code ontology and documentation ontology are linked, queries regarding the source code entities, design level concepts, and their occurrences in documents can be performed using the reasoning services provided by our ontology reasoner, Racer. For example, during the comprehension of the class `FeatureStore`, a reverse engineer may want to study the classes that are related to `FeatureStore`. Within the source code ontology, a query can be performed to retrieve all classes that contain methods that have called class `FeatureStore`. In addition to these types of source code queries, the reverse engineer can perform queries that cross the boundaries between source code and documentation. Such type of queries are enabled due to the already established links between the source code and documentation ontology.

The linked source code and documentation ontologies also provide us with the capability to combine semantic information from both software implementation and documentation. For example, our text mining system has detected that class `AST` is potentially a part of a Visitor pattern (Figure 14). In order to retrieve all documented information related to the detected pattern, the following query can be used to retrieve all text paragraphs that describe the sub-classes of `AST`:

```

1 var query = new Query();           // define a new query
2 query.declare( "P", "C" );       // declare two query variables
3 query.restrict( "P", "Paragraph" ); // P is a paragraph
4 query.restrict( "C", "Class" );  // C is a class
5 query.restrict( "C", "hasSuper", // C is a sub-class of AST
6   "net.refractions.udig.catalog.util.AST" );
7 query.restrict( "P", "contains", "C" ); // P contains C
8 query.retrieve( "P" );           // this query only retrieves P
9 var result = ontology.query(query); // perform the query

```

This query combines two parts of the ontology: first, the programming language semantics, such as the inheritance relation between query variable `C` and the class `AST`, and second, the structural information of documentation, such as the containing relation between a paragraph `P` and the class `C`. The result of this query therefore contains all text paragraphs that describe the sub classes of `AST`, i.e., the *Visitor* pattern. It has to be noted that the role *contains* is a transitive relation to describe the document structure. The ontology reasoner can automatically resolve the transitivity from Paragraph to Sentence, and from Sentence to Class.

Summary. In this section, we presented an initial evaluation of recovering traceability links between source code and documentation on a large open source software system. We have demonstrated the use of automated reasoning to retrieve documented information with regard to a specific reverse engineering task and infer implicit relations in the linked ontologies.

6 Related Work and Discussion

In this section, we present work similar to ours, structured into three areas: (i) applications of NLP to software documents, (ii) other approaches for recovering

traceability links between code and documentation; and (iii) ontological approaches to the software engineering domain.

NLP for Software Documentation. Very little previous work exists on text mining software documents. Most of this research has focused on analysing texts at the specification level, e.g., in order to automatically convert use case descriptions into a formal representation [11, 20] or detect inconsistent requirements [14]. In contrast, we aim to support the complete software documentation life-cycle, from white papers, design and implementation documents to in-line code texts (e.g., JavaDoc). To the best of our knowledge, there has been so far no attempt to automatically cross-link entities (e.g., methods, design patterns, architectures) detected by text mining software documents with corresponding entities found by source code analysis, which is an important contribution of our work. Likewise, we are not aware of previous approaches that include knowledge derived from analysing source code into the natural language analysis of the corresponding documentation, which we believe has potential far beyond the first experiments described in this paper.

Code/Document Traceability. There exists some research in recovering traceability links between source code and design documents using Information Retrieval techniques. The IR models used include traditional vector space and probabilistic models [1, 2], as well as latent semantic indexing (LSI) [18, 19]. The approach by [2] applies both a probabilistic and a vector space information retrieval model in two case studies to trace C++ source code onto manual pages and Java code to functional requirements. The approach in [18] uses LSI to extract the meaning (semantics) of the documentation and source code, and then use this information to identify traceability links based on similarity measures.

In contrast with these IR approaches, our work also takes advantage of structural and semantic information in both the documentation and the source code by means of text mining and source code parsing. The resulting analysis is much more fine-grained than IR approaches, identifying individual words and phrases in a document. IR approaches, like the one by Antoniol et al. [2], work by linking a class to a whole manual page, whereas we can link a class to its precise occurrence within a page, including those occurrences where it is only mentioned by a pronoun. The same applies to the LSI approach by Marcus et al. [19], who generate links for whole documents or document parts (e.g.,

sections), not individual words or phrases like our system. Likewise, links are established with much larger structures on the source code side—files or classes, not individual methods or variables. As the results of these methods are unstructured clusters, advanced queries and reasoning are also not possible.

Software Ontologies Ontologies as a formal knowledge representation mechanism have been applied to many areas in software engineering [6]. Common to all of these approaches is that their main intent is to support in one form or another the conceptualization of knowledge, mainly by standardization of terminology, and to support knowledge sharing based on a common understanding. These approaches fall short on adopting and implementing automated population tools to support analysis tasks using the ontology as a knowledge base. They also lack the use of reasoning services to infer implicit knowledge. Finally, there is some work targeting ontology learning from software documentation. Our work differs in that we construct the ontology specifically for text mining, whereas the work by Sabou [25] applies text mining for ontology learning. We believe that current methods for automated ontology construction do not provide the necessary level of detail as discussed in Section 2.3. However, a future combination of both approaches might be a promising target of research.

7 Conclusions and Future Work

We presented a text mining system for the software domain that is capable of extracting entities from software documents. The system's output is a populated OWL-DL ontology containing normalized instances and their relations. The system is novel in two important aspects: First, it employs a formal ontology, based on description logics, both as a processing resource for the various NLP components and the result export format. Second, as the system is part of a larger ontology-based program comprehension environment, it can incorporate results from automated source code analysis subsystems in its NLP processing pipeline.

The ontological foundation allows for important improvements in software engineering, as it supports *queries* and *reasoning* services on semantic knowledge automatically derived from large amounts of documentation in natural language form. We previously showed how automated reasoning can support a software maintainer when performing knowledge-intensive tasks, like architectural recovery or source code security analy-

sis [34]. We are also currently experimenting with ontology alignment strategies to automatically detect inconsistencies between code and its corresponding documentation.

The ontological representation allows to perform high-level software analysis tasks that include both code and documents within the same data structure. One obvious application area is the recovery of traceability links, as shown in Figure 14. This allows, for the first time, the automatic establishment and analysis of semantic links, down to the level of individual words in documents and variables in code, which is of high importance for the software industry. Here, we see the systematic evaluation of automatic traceability analysis systems as the next major step to be carried out by the community. A possible approach is the definition of a yearly “competition,” with shared data, tasks, and evaluation metrics for all participants, similar to the ones sponsored by the U.S. NIST within the fields of text retrieval (TREC)¹⁶ and automatic summarization (DUC).¹⁷ Possible tasks could include the automatic recovery of traceability links between source code and documents, or the analysis of existing links for inconsistencies. To allow the automatic evaluation of system results, a manual “gold standard” would need to be developed, together with suitable performance metrics (e.g., adapted versions of the common precision/recall measures). Such competitions are routinely used within the field of NLP to assess the state of the art within a given task, as well as for the evaluation of scientific progress achieved by new methods. The experience from these competitions show that, although involving a major community-wide investment, the gained insights for a given research problem is well worth the effort.

Besides improving the individual components as discussed in the evaluation section, we plan to extend our system to explicitly deal with documents associated with the different steps in the software life-cycle, from white papers and requirements over design and implementation documents to user’s guides and source code comments. This will allow us to trace concepts and entities across the different states of software development and different levels of abstraction.

References

- [1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Information re-

trieval models for recovering traceability links between code and documentation. In *Proc. of IEEE Intl. Conf. on Software Maintenance*, San Jose, CA, USA, 2000.

- [2] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
- [3] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.
- [4] Franz Baader, Diego Calvanese, Deborah L. MacGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, second edition, 2007.
- [5] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Limited, 1999.
- [6] Coral Calero, Francisco Ruiz, and Mario Piattini, editors. *Ontologies for Software Engineering and Software Technology*. Springer-Verlag Berlin Heidelberg, 2006.
- [7] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proc. of the 40th Anniversary Meeting of the ACL*, 2002.
- [8] R. Gaizauskas, M. Hepple, H. Saggion, M. A. Greenwood, and K. Humphreys. SUPPLE: A practical parser for natural language engineering applications. In *Proc. of the 9th Intl. Workshop on Parsing Technologies (IWPT2005)*, Vancouver, 2005.
- [9] Volker Haarslev and Ralf Möller. RACER System Description. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR)*, pages 701–705, Siena, Italy, June 18–23 2001. Springer-Verlag Berlin.
- [10] IEEE. IEEE Standard for Software Maintenance. IEEE 1219, 1998.
- [11] M.G. Ilieva and O. Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *10th International Conference on Applications of Natural Language to Information Systems (NLDB)*, volume 3513 of *LNCS*, pages 392–397, Alicante, Spain, June 15–17 2005. Springer.
- [12] D. Jin and J. Cordy. Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology. In *21st*

¹⁶Text REtrieval Conference (TREC), <http://trec.nist.gov/>

¹⁷Document Understanding Conference (DUC), <http://duc.nist.gov>

- IEEE International Conference on Software Maintenance (ICSM)*, 2005.
- [13] P. N. Johnson-Laird. *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness*. Harvard University, Cambridge, Mass., 1983.
- [14] Leonid Kof. Natural language processing: Mature enough for requirements documents analysis? In *10th International Conference on Applications of Natural Language to Information Systems (NLDB)*, volume 3513 of *LNCS*, pages 91–102, Alicante, Spain, June 15–17 2005. Springer.
- [15] T. C. Lethbridge and A. Nicholas. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. Technical Report TR-97-07, Department of Computer Science, University of Ottawa, 1997.
- [16] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *Journal of Systems and Software*, 43(1):19–27, 1998.
- [17] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
- [18] Andrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proc. of 25th Intl. Conf. on Software Engineering*, 2002.
- [19] Andrian Marcus, Jonathan I. Maletic, and Andrey Sergeyev. Recovery of Traceability Links between Software Documentation and Source Code. *Intl. J. of Software Engineering and Knowledge Engineering*, 15(5):811–836, 2005.
- [20] Vladimir Mencl. Deriving behavior specifications from textual use cases. In *Proceedings of Workshop on Intelligent Technologies for Software Engineering*, pages 331–341, Linz, Austria, 2004. Oesterreichische Computer Gesellschaft.
- [21] W. Meng, J. Rilling, Y. Zhang, R. Witte, and P. Charland. An Ontological Software Comprehension Process Model. In *3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, pages 28–35, Genoa, Italy, October 1st 2006.
- [22] N. F. Noy and H. Stuckenschmidt. Ontology Alignment: An annotated Bibliography. In *Semantic Interoperability and Integration*, Schloss Dagstuhl, Germany, 2005.
- [23] Juergen Rilling, René Witte, and Yonggang Zhang. Automatic Traceability Recovery: An Ontological Approach. In *International Symposium on Grand Challenges in Traceability (GCT'07)*, Lexington, Kentucky, USA, March 22–23 2007.
- [24] C. Riva. Reverse Architecting: An Industrial Experience Report. In *7th IEEE Working Conference on Reverse Engineering (WCRE)*, pages 42–52, 2000.
- [25] Marta Sabou. Extracting Ontologies from Software Documentation: a Semi-Automatic Method and its Evaluation. In *ECAI-2004 Workshop on Ontology Learning and Population*, Valencia, Spain, 2004.
- [26] R. Seacord, D. Plakosh, and G. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI Series in SE. Addison-Wesley, 2003.
- [27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [28] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2000.
- [29] M. A. Storey, S. E. Sim, and K. Wong. A Collaborative Demonstration of Reverse Engineering tools. *ACM SIGAPP Applied Computing Review*, 10(1):18–25, 2002.
- [30] C. Welty. Augmenting Abstract Syntax Trees for Program Understanding. In *Proc. of Int. Conf. on Automated Software Engineering*, pages 126–133. IEEE Comp. Soc. Press, 1997.
- [31] René Witte and Sabine Bergler. Fuzzy Coreference Resolution for Summarization. In *Proceedings of 2003 International Symposium on Reference Resolution and Its Applications to Question Answering and Summarization (ARQAS)*, pages 43–50, Venice, Italy, June 23–24 2003. Università Ca' Foscari.
- [32] René Witte and Sabine Bergler. Next-Generation Summarization: Contrastive, Focused, and Update Summaries. In *International Conference on Recent Advances in Natural Language Processing (RANLP 2007)*, Borovets, Bulgaria, September 27–29 2007.
- [33] René Witte, Thomas Kappler, and Christopher J. O. Baker. Ontology Design for Biomedical Text Mining. In *Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences*, chapter 13, pages 281–313. Springer, 2007.
- [34] René Witte, Yonggang Zhang, and Juergen Rilling. Empowering Software Maintainers with Semantic Web Technologies. In E. Franconi, M. Kifer, and W. May, editors, *4th European Semantic Web Conference (ESWC 2007)*, number 4519 in *LNCS*, pages 37–52, Innsbruck, Austria, June 2007. Springer-Verlag Berlin Heidelberg.