Empowering Software Maintainers with Semantic Web Technologies

René Witte[1], Yonggang Zhang[2], and Jürgen Rilling[2]

[1] Institute for Progam Structures and
Data Organisation (IPD), Faculty of Informatics
University of Karlsruhe, Germany
witte@ipd.uka.de
[2] Department of Computer Science
and Software Engineering
Concordia University, Montreal, Canada
{rilling,yongg_zh}@cse.concordia.ca

Abstract. Software maintainers routinely have to deal with a multitude of artifacts, like source code or documents, which often end up disconnected, due to their different representations and the size and complexity of legacy systems. One of the main challenges in software maintenance is to establish and maintain the semantic connections among all the different artifacts. In this paper, we show how Semantic Web technologies can deliver a unified representation to explore, query and reason about a multitude of software artifacts. A novel feature is the automatic integration of two important types of software maintenance artifacts, source code and documents, by populating their corresponding sub-ontologies through code analysis and text mining. We demonstrate how the resulting "Software Semantic Web" can support typical maintenance tasks through ontology queries and Description Logic reasoning, such as security analysis, architectural evolution, and traceability recovery between code and documents.

Keywords: Software Maintenance, Ontology Population, Text Mining.

1 Introduction and Motivation

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 [15, 16]. Software maintenance is a multi-dimensional problem space that creates an ongoing challenge for both the research community and tool developers [8,14]. These maintenance challenges are caused by the different representations and interrelationships that exist among software artifacts and knowledge resources [17,18]. From a maintainer's perspective, exploring [11] 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 [13].

E. Franconi, M. Kifer, and W. May (Eds.): ESWC 2007, LNCS 4519, pp. 37–52, 2007.

In this research, we introduce a novel formal ontological representation that integrates two of the major software artifacts, source code and software documentation, thereby reducing the conceptual gap between these artifacts. Discovered concepts and concept instances from both source code and documents are used to explore and establish the links between these artifacts, providing maintainers with support during typical software maintenance tasks [2].

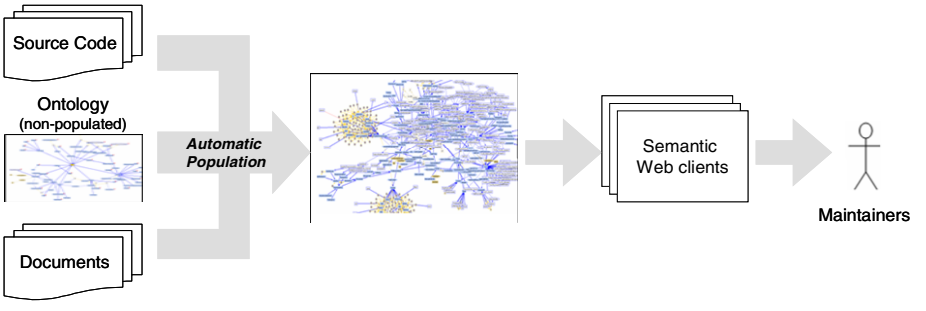

Fig. 1. Ontology-Based Software Maintenance Overview

A general overview of our approach is shown in Fig. 1. In a first step, the existing ontology is automatically populated from both the source code and documentation artifacts. In a second step, the resulting knowledge base is explored, queried and reasoned upon by utilizing various Semantic Web-enabled clients.

Our research is significant for several reasons: *(1)* The fact that we provide a novel approach to unify different software artifacts using a Semantic Web approach. *(2)* We developed fully automatic ontology population that allows us to take advantage of the large body of existing software artifacts, namely software documents and source code and the knowledge they contain. *(3)* We present concrete application examples, illustrating how our ontological representation can benefit software developers during typical maintenance tasks.

In Section 2, we discuss both challenges and requirements for a Semantic Web approach to software maintenance. Section 3 provides a general overview of our system. The design for our software and source code ontologies is discussed in Section 4. In Section 5, we describe in detail the fully automatic population of the ontologies, followed by concrete examples illustrating how our approach can benefit software engineers during typical maintenance tasks in Section 6.

2 Semantic Web and Software Maintenance

In a complex application domain like software maintenance, knowledge needs to be continually integrated from different sources (like source code repositories, documentation, test case results), different levels of scope (from single variables to complete system architectures), and across different relations (static, dynamic, etc.) [7, 9]. No single system is currently capable of supporting a complete domain like software engineering by itself. This makes it necessary to develop focused applications that can

deal with individual aspects in a reliable manner, while still being able to integrate their results into a common knowledge base. Ontologies offer this capability: a large body of work exists that deals with ontology alignment and the development of upper level ontologies, while Description Logic (DL) reasoners can check the internal consistency of a knowledge base, ensuring at least some level of semantic integrity. However, before we can design Semantic Web support for software maintenance, we have to analyze the requirements particular to that domain.

2.1 Software Maintenance Challenges

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 [16]. The relative cost of maintaining and managing the evolution of this large software base represents now more than 90% of the total cost [15] associated with a software product. One of the major challenges for the maintainers 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 [9]. These artifacts include, among others, source code and software documents (e.g., requirements, design documentation). From a maintainer's perspective, it becomes essential to establish and maintain the semantic connections among these artifacts.

In what follows, we introduce three typical use cases, which we will later revisit to illustrate the applicability of our approach in supporting software maintainers during these typical maintenance tasks.

Use case #1: *Identify security concerns in source code.* As discussed in [11], source code searching and browsing are two of the most common activities during the maintenance of existing software. With applications that become exposed to volatile environments with increased security risks (e.g., distributed environments, web-centric applications), identifying these security flaws in existing software systems becomes one of the major activities in the software maintenance phase.

Use case #2: *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 [12]. 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.

Use case #3: *Architectural recovery and restructuring.* With their increasing size and complexity, maintaining the overall structure of software systems becomes an emerging challenge of software maintenance. Maintainers need to comprehend the overall structure of a software system by identifying major components and their properties, as well as linking identified components with their lower-level implementation [14].

2.2 Identified Requirements

Based on the stated use cases, we can now derive requirements for Semantic Web support of software maintenance.

As a prerequisite, a sufficiently large part of the domain must be modeled in form of an ontology, including the structure and semantics of source code and documents to a level of detail that allows relevant queries and reasoning on the properties of existing artifacts (e.g., for security analysis).

Software maintenance intrinsically needs to deal with a large number of artifacts from legacy systems. It is not feasible to manually create instance information for existing source code or documents due to the large number of concept instances that exist in these artifacts. Thus, automatic ontology population methods must be provided for extracting semantic information from those artifacts.

The semantic information must be accessible through a software maintainer's desktop. Knowledge obtained through querying and reasoning should be integrated with existing development tools (e.g., *Eclipse*).

Finally, the acceptance of Semantic Web technologies by software maintainers is directly dependent on delivering added benefits, specifically improving on typical tasks, such as the ones described by the use cases above.

3 System Architecture and Implementation

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 and their relations from source code and documents (Fig. 2).

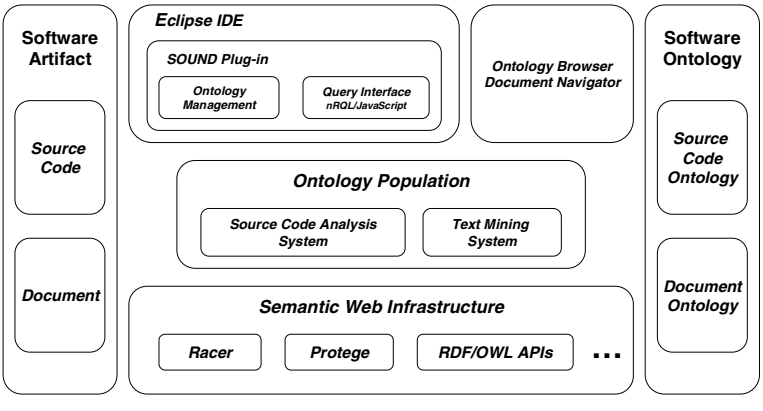

Fig. 2. Semantic Web-enabled Software Maintenance Architecture

An important part of our architecture is a *software ontology* that captures major concepts and relations in the software maintenance domain [6]. This ontology consists of two sub-ontologies: a *source code* and *document* ontology, which represent

information extracted from source code and documents, respectively. The ontologies are modeled in OWL-DL[1] and were created using the Protégé-OWL extension of Protégé,[2] a free ontology editor.

Racer [5], an ontology inference engine, is adopted to provide reasoning services. The Racer system is a highly optimized DL system that supports reasoning about instances, which is particularly useful for the software maintance domain, where a large amount of instances needs to be handled efficiently.

Automatic ontology population is handled by two subsystems: The source code analysis, which is based on the JDT Java parser[3] provided by Eclipse[4]; and the document analysis, which is a text mining system based on the GATE (*General Architecture for Text Engineering)* framework [3].

The query interface of our system is a plug-in that provides OWL integration for Eclipse, a widely used software development platform. 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[5]. This language simplifies querying the ontology for software engineers not familiar with DL-based formalisms.

4 Ontology Design for Software Maintenance

Software artifacts, such as source code or documentation, typically contain knowledge that is rich in both structural and semantic information. Providing a uniform ontological representation for various software artifacts enables us to utilize semantic information conveyed by these artifacts and to establish their traceability links at the semantic level. In what follows, we discuss design issues for both the documentation and source code ontology used in our approach.

4.1 Source Code Ontology

The source code ontology has been designed to formally specify major concepts of object-oriented programming languages. In our implementation, this ontology is further extended with additional concepts and properties needed for some specific languages (in our case, Java). Examples for classes in the source code ontology are Package, Class, Method, or Variable. Our source code ontology is described in more detail in [20].

Within this sub-ontology, various ObjectProperties are defined to characterize the relationships among concepts. For example, two instances of SourceObject may have a definedIn relation indicating one is defined in the other; or an instance of method may read an instance of Field indicating the method may read the field in the body of the method.

---

[1] OWL Web Ontology Language Guide, W3C, http://www.w3.org/TR/owl-guide/
[2] Protégé ontology editor, http://protege.stanford.edu/
[3] Eclipse Java Development Tools (JDT), http://www.eclipse.org/jdt/
[4] Eclipse, http://www.eclipse.org
[5] Rhino JavaScript interpreter, http://www.mozilla.org/rhino/

Concepts in the source code ontology typically have a direct mapping to source code entities and can therefore be automatically populated through source code analysis (see Section 5.1).

4.2 Documentation Ontology

The documentation ontology 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 outlined in [19] for the software engineering domain. Specifically, we included:

A *Text Model* to represent the structure of documents, e.g., classes 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.

Lexical Information facilitating the detection of entities in documents, like the names of common design patterns, programming language-specific keywords, or architectural styles; and lexical normalization rules for entity normalization.

Relations between the classes, which include the ones modeled in the source code ontology. These allow us to automatically restrict NLP-detected relations to semantically valid ones (e.g., 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).

Finally, *Source Code Entities* that have been automatically populated through source code analysis (cf. Section 5.1) can also be utilized for detecting corresponding entities in documents, as we describe in more detail in Sections 5.2.

5 Automatic Ontology Population

One of the major challenges for software maintainers is the large amount of information that has to be explored and analyzed as part of typical maintenance activities. Therefore, support for automatic ontology population is essential for the successful adoption of Semantic Web technology in software maintenance. In this section, we describe in detail the automatic population of our ontologies from existing artifacts: source code (Section 5.1) and documents (Section 5.2).

5.1 Populating the Source Code Ontology

The source code ontology population subsystem is based on JDT, 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 (Figure 3).

As an example, consider a single line of Java source code: *public int sort(){*, which declares a method called ***sort***. A simplified AST corresponding to this line of source code is shown in Fig. 3. We traverse this tree by first visiting the root node *Method Declaration*. At this step, the system understands that a *Method* instance shall be created. Next, the *Name Node* is visited to create the instance of the *Method* class, in this case ***sort***. Then the *Modifier Node* and *Type Node* are also visited, in order to establish the relations with the identified instance. As a result, two relations, ***sort*** *hasModifier* ***public*** and ***sort*** *hasType* ***int,*** are detected.

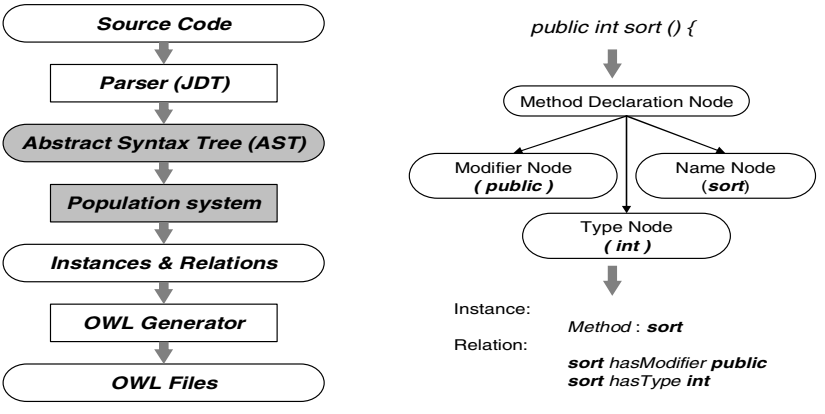

Fig. 3. Populating the source code ontology

The numbers of instances and relations identified by our system depend on the complexity of the ontology and the size of the source code to be analyzed. At the current stage of our research, the source code ontology contains 38 concepts (classes) and 41 types of relations (ObjectProperties). We have performed several case studies on different open source systems to evaluate the size of the populated ontology. Table 1 summaries the results of our case studies, with the size of the software system being measured by lines of code (LOC) and the process time reflecting both AST traversal and ontology population.

Table 1. Source code Ontology size for different open source projects

	LOC	Proc. Time	Instances	Relations	Inst./LOC	Rel./LOC
java.util	24k	13.62s	10140	47009	0.42	1.96
InfoGlue[6]	40k	27.61s	15942	77417	0.40	1.94
Debrief[7]	140k	67.12s	52406	244403	0.37	1.75
uDig[8]	177k	82.26s	69627	284692	0.39	1.61

[6] Infoglue, http://www.infoglue.org
[7] Debrief, http://www.debrief.info
[8] uDig, http://udig.refractions.net

5.2 Populating the Documentation Ontology

We developed a custom text mining system to extract knowledge from software documents and populate the corresponding sub-ontology. The processing pipeline and its connection with the software documentation sub-ontology is shown in Fig. 4. Note that, in addition to the software documentation ontology, the text mining system can also import the instantiated source code ontology corresponding to the document(s) under analysis.

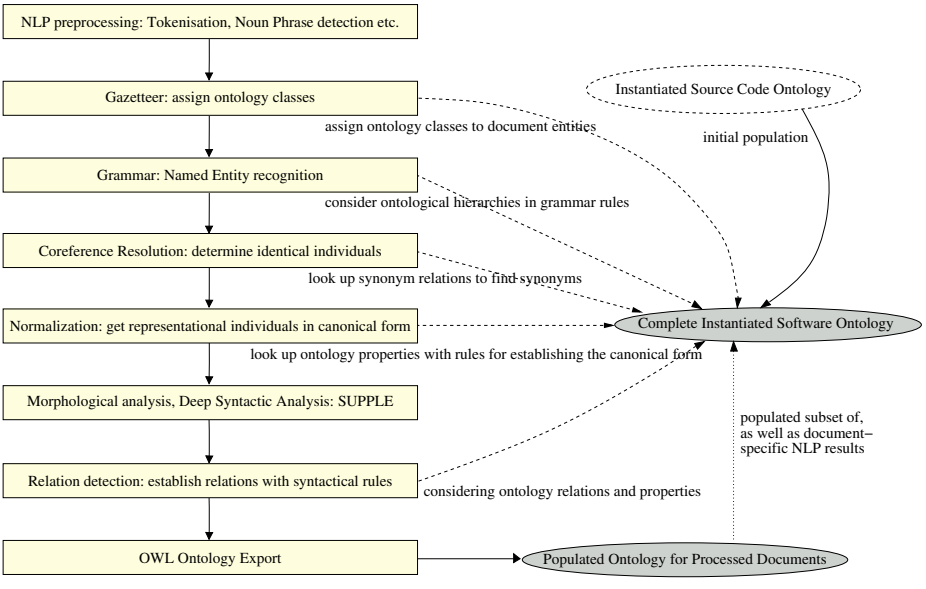

Fig. 4. Workflow of the Ontology-Driven Text Mining Subsystem

The system first performs a number of standard preprocessing steps, such as tokenisation, sentence splitting, part-of-speech tagging and noun phrase chunking.[9] Then, named entities (NEs) modeled in the software ontology are detected in a two-step process: Firstly, an *OntoGazetteer* is used to annotate tokens with the corresponding class or classes in the software ontology (e.g., the word "architecture" would be labeled with the *architecture* class in the ontology). Complex named entities are then detected in the second step using a cascade of finite-state transducers implementing custom grammar rules written in the JAPE language, which is part of GATE. These rules refer back to the annotations generated by the OntoGazetteer, and also evaluate the ontology. For example, in a comparison like `class=="Keyword"`, the ontological hierarchy is taken into account so that a `JavaKeyword` also matches, since a Java keyword *is-a* keyword in the ontology. This significantly reduces the overhead for grammar development and testing.

[9] For more details, please refer to the GATE documentation: http://gate.ac.uk/documentation/

The next major steps are the *normalization* of the detected entities and the resolution of *co-references*. Normalization computes a canonical name for each detected entity, which is important for automatic ontology population. In natural language texts, an entity like a method is typically referred to with a phrase like *"the myTestMethod provides..."*. Here, only the entity *myTestMethod* should become an instance of the *Method* class in the ontology. This is automatically achieved through lexical normalization rules, which are stored in the software ontology as well, together with their respective classes. Moreover, throughout a document a single entity is usually referred to with different textual descriptors, including pronominal references (like *"this method")*. In order to find these references and export only a single instance into the ontology that references all these occurrances, we perform an additional co-reference resolution step to detect both nominal and pronomial coreferences.

The next step is the detection of *relations* between the identified entities in order to compute predicate-argument structures, like *implements(class, interface)*. Here, we combine two different and largely complementary approaches: A deep syntactic analysis using the SUPPLE bottom-up parser and a number of pre-defined JAPE grammar rules, which are again stored in the ontology together with the relation concepts.

Finally, the text mining results are exported by populating the software documentation sub-ontology using a custom GATE component, the *OwlExporter*. The exported, populated ontology also contains document-specific information; for example, for each class instance the sentence it was found in is recorded. Figures 5 and 6 show excerpts of ontologies populated by our text mining system.

6 Application of Semantic Web-Enabled Software Maintenance

In what follows, we describe concrete application scenarios that correspond to the three use cases introduced earlier in Section 2.1.

6.1 Source Code Security Analysis

Existing techniques on detecting and correcting software security vulnerabilities at the source code level include human code reviews, testing, and static analysis. In the following example, we illustrate how our Semantic Web-based approach can facilitate security experts or programmers in identifying potential vulnerabilities caused by unexpected object accessibility.

In this scenario, a maintainer may consider allowing public and non-final fields in Java source code a security risk that may cause the value of the field being modified outside of the class where it was defined. In order to detect this, he can search the ontology through a query[10] that retrieves all `Field` instances that have a `PublicModifier` but no `FinalModifier`:

```
var SecurityConcern1 = new Query();                    // define a new query
SecurityConcern1.declare("F", "MP", "MF" );            // declare three query variables
SecurityConcern1.restrict("F", "Field");               // variable F must be a Field instance
SecurityConcern1.restrict("MP", "PublicModifier");     // variable MP must be a PublicModifier instance
```

[10] In this and the following examples, we present ontology queries using our JavaScript-based query interface discussed in Section 3.

```
SecurityConcern1.restrict("MF", "FinalModifier");          // variable MF must be a FinalModifier instance
SecurityConcern1.restrict("F", "hasModifier", "MP");       // F and MP have a hasModifier relation
SecurityConcern1.no_relation("F", "hasModifier", "MF");    // F and MF have NO hasModifier relation
SecurityConcern1.retrieve("F");                            // this query only retrieve F
var result = ontology.query(SecurityConcern1);             // perform the query
```

In order to extend the query for more specific tasks, such as: *Retrieve all public data of Java package "user.pkg1" that may potentially be (read or write) accessed by a package "user.pkg2"*, the previous query can be further refined by adding:

```
SecurityConcern1.restrict("F", "definedIn", "user.pkg1");  // F must be definedIn user.pkg1
SecurityConcern1.restrict("M", "Method");                  // variable M must be a Method instance
SecurityConcern1.restrict("M", "definedIn", "user.pkg2");  // M must be definedIn user.pkg2
SecurityConcern1.restrict("M", "access", "F");             // M and F have an access relation
```

It should be noted that fields or methods in Java are defined in classes, and classes are defined in packages. The ontology reasoner will automatically determine the transitive relation definedIn between the concepts Field/Method and Package. In addition, read and write relations between method and field are modeled in our ontology by the readField and writeField ObjectProperties, which are a subPropertyOf access.

Many security flaws are preventable through security enforcement. Common vulnerabilities such as buffer overflows, accessing un-initialized variables, or leaving temporary files in the disk could be avoided by programmers with strong awareness of security concerns. In order to deliver more secure software, many development teams have guidelines for coding practice to enforce security. In our approach, we support maintainers and security experts during enforcement or validation, by checking whether these programming guidelines are followed. For example, to prevent access to un-initialized variables, a general guideline could be: *all fields must be initialized in the constructors.* The following query retrieves all classes that did not follow this specific constructor initialization guideline:

```
var SecurityConcern2 = new Query();                        // define a new query
SecurityConcern2.declare("F", "I", "C");                   // declare three query variables
SecurityConcern2.restrict("F", "Field");                   // variable F must be a Field instance
SecurityConcern2.restrict("I", "Constructor");             // variable I must be a Constructor instance
SecurityConcern2.restrict("C", "Class");                   // variable C must be a Class instance
SecurityConcern2.restrict("F", "definedIn",  "C");         // F must be definedIn C
SecurityConcern2.restrict("I", "definedIn", "C");          // I must be also definedIn C
SecurityConcern2.no_relation("I", "writeField", "F");      // I and F have NO writeField relation
SecurityConcern2.retrieve("C", "I");                       // this query only retrieve C and I
var result = ontology.query(SecurityConcern2);             // perform the query
```

These two examples illustrate the power of our Semantic Web-enabled software maintenance approach: Complex queries can be performed on the populated ontology to identify specific patterns in the source code. Such types of queries utilize both the structural (e.g., definedIn) and semantic (e.g., writeField) knowledge of programming languages, which is typically ignored by traditional search tools based on string-matching, such as *grep*[11].

6.2 Establishing Traceability Links Between Source Code and Documentation

After instantiating both the source code and documentation sub-ontologies from their respective artifacts, it is now possible to automatically cross-link instances between

[11] Grep tool, http://www.gnu.org/software/grep/

these sub-ontologies. This allows maintainers to establish traceability links among the sub-ontologies through queries and reasoning, in order to find, for example, documentation corresponding to a source code entity, or to detect inconsistencies between information contained in natural language texts vs. the actual code.

For example, our source code analysis tool may identify c_1 and c_2 as classes; and this information can be used by the text mining system to identify named entities – c'_1 and c'_2 – and their associated information in the documents (Fig. 5). As a result, source code entities c_1 and c_2 can now be linked to their occurrences in the documents (c'_1 and c'_2). After source code and documentation ontology are linked, users can perform ontological queries on either documents or source code regarding properties of c_1 or c_2. For example, in order to retrieve document passages that describe both c_1 and c_2 or to retrieve design pattern descriptions referring to a pattern that contains the class currently being analyzed by a maintainer. Furthermore, it is also possible to identify inconsistencies – the documentation might list a method as belonging to a different class than it is actually implemented, for example – which are detected through the linking process and registered for further review by the user.

We performed an initial evaluation on a large open source Geographic Information System (GIS), uDig[12], which is implemented as a set of plug-ins on top of the Eclipse platform. The uDig documents used in the study consist of a set of JavaDoc files and a requirement analysis document.[13]

Links between the uDig implementation and its documentation are recovered by first performing source code analysis to populate the source code ontology. The resulted ontology contains instances of Class, Method, Field, etc., and their relations, such as inheritance and invocation. Our text mining system takes these identified class names, method names, and field names as an additional resource to populate the documentation ontology (cf. Fig. 4). Through this text mining process, a large number of Java language concept instances are discovered in the documents, as well as design-level concept instances such as design patterns or architecture styles. Ontology linking rules are then applied to link the populated documentation and source code ontologies.

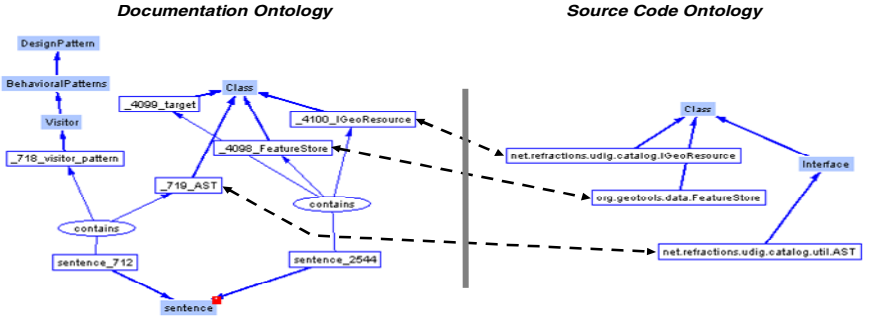

Fig. 5. Linked Source Code and Documentation Ontology

[12] uDIG open source GIS, http://udig.refractions.net/confluence/display/UDIG/Home
[13] uDig documentation, http://udig.refractions.net/docs/

A partial view of a linked ontology is shown in Figure 5; the corresponding sentences are:

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 5 shows that our text mining system was able to discover that *sentence_2544* contains both class instances *_4098_FeatureStore* and *_4100_IGeoResource*. Both of these classes can be linked to the instances in source code ontology, *org.geotools.data.FeatureStore* and *net.refractions.udig.catalog.IGeoResource*, respectively. Additionally, in *sentence_712*, a class instance (*_719_AST*) and a design pattern instance (*_718_visitor_pattern*) are also identified. Instance *_719_AST* is linked in a similar manner to the *net.refractions.udig.catalog.util.AST* interface in the source code ontology. Therefore, the recovery of traceability links between source code and documentation is feasible and implicit relations in the linked ontologies can be inferred.

6.3 Architectural Analysis

The populated ontology can also assist maintainers in performing more challenging tasks, such as analyzing the overall structure of a software system, i.e., architectural analysis. In this case study, we analyzed the architecture of the open source web site content management system, *InfoGlue*[14]. The first step of an architectural analysis is wtypically to identify potential architectural styles [7] and candidate components in the system. By browsing the documentation ontology populated through text mining, we observe that a large number of instances of concept Layer are discovered. This information provides us with significant clues that the InfoGlue system might be implemented using a typical *Layered Architecture* [7]. Additionally, the text mining discovered that the application layer contains a set of action classes, as shown in Fig. 6. This information provides important references for our further analysis of the documents and source code.

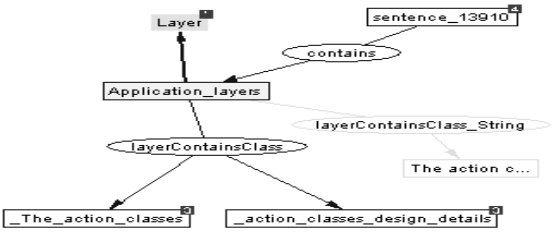

Fig. 6. Architecture information discovered by text mining

We later determined that the action classes refer to classes that implement *webwork.action.Action* interface. Before conducting the analysis, we hypothesized

[14] InfoGlue Open Source Content Management Platform, http://www.infoglue.org/

that the InfoGlue system implements a common layered architecture, in which each layer only communicates with its upper or lower layer. In order to validate our hypothesis, we performed a number of queries on the populated source code ontology to retrieve method calls between layers.

The script first retrieves all layer instances in the ontology, and then iteratively queries method call relations between layers. A similar query is performed to retrieve the number of methods being called.

```
var layers = ontology.retrieve_instance("Layer");

for(var i = 0; i < layers.size(); i++){
    var layer1 = layers.get("Layer", i);
    for(var j = 0; j < layers.size(); j++){
        var layer2 = layers.get("Layer", j);
        if(layer1.equals(layer2)) continue;
        var query = new Query();
        query.declare("M1", "M2");
        query.restrict("M1", "Method");
        query.restrict("M2", "Method");
        query.restrict("M1", "definedIn", layer1);
        query.restrict("M2", "definedIn", layer2);
        query.restrict("M1", "call", "M2");
        query.retrieve("M1", "M2");
        var result = ontology.query(query);
        out.println(layer1 + " calls " + layer2 + " " + result.size() + " times.");
    }
}
```

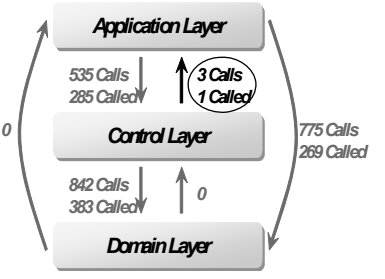

Fig. 7. Example script to detect method calls between layers (left) and results obtained from executing the query on the populated ontology (right)

Fig. 7 summarizes the results of these two queries, by showing both the number of method calls and the number of methods being called. From the analysis of the result one can refute the original hypothesis about the implementation of the common layered architecture. This is due to the fact that one can observe in the InfolGlue system a significant amount of communications from the application layer to domain layer – skipping the control layer. This information is valuable for software maintainers, because it indicates that any changes made in the domain layer may also directly affect the application layer, a situation which one would not expect based on the architectural description found in the InfoGlue system documentation.

In addition, we observed that there is no communication from the domain layer to the control and application layer, i.e., the domain layer can be substituted by other components matching the same interface. This observation also reveals an important property of the domain layer in the InfoGlue system – the domain layer is a self-contained component that can be reused by other applications. Our observation is also supported by the architecture document itself, which clearly states that "*the domain business logic should reside in the domain objects themselves making them self contained and reusable*".

Moreover, by analyzing these results, one would expect that a lower layer should not communicate with its upper layer. The three method calls from the control layer to the application layer can therefore be considered as either implementation defects or as the result of a special design intention not documented. Our further inspection showed that

the method being called is a utility method that is used to format HTML content. We consider this to be an implementation defect since the method can be re-implemented in the control layer to maintain the integrity of a common layered architecture style.

7 Related Work and Discussions

Existing research on applying Semantic Web techniques in software maintenance mainly focuses on providing ontological representation for particular software artifacts or supporting specific maintenance task [10]. In [21], Ankolekar et al. provide an ontology to model software, developers, and bugs. This ontology is semi-automatically populated from existing artifacts, such as software interface, emails, etc. Their approach assists the communication between software developers for bug resolution. In [22], Happle et al. present an approach addressing the component reuse issue of software development by storing descriptions of components in a Semantic Web repository, which can then be queried for existing components.

Comparing with the existing approaches, like the LaSSIE system [4], our work differs in two important aspects: *(1)* the automatic population from existing software artifacts, especially source code and its documentation, which are both very different in structure and semantics; and *(2)* the application of queries on the populated ontologies, including DL reasoning, to enhance concrete tasks performed by software maintainers. The first aspect is an important prerequisite to bring a large amount of existing data into the "Software Semantic Web". The inclusion of semantically different and complementary artifacts, in the form of machine-readable code and natural language, provides for real improvement in software maintenance, enabling for the first time an automatic connection between code and its documentation. The second aspect shows the power of DL-based reasoning when applied to the software domain, significantly enhancing the power of conventional software development tools.

8 Conclusions and Future Work

In this paper, we presented a novel approach that provides formal ontological representations of the software domain for both source code and document artifacts. The ontologies capture structural and semantic information conveyed in these artifacts, and therefore allow us to link, query and reason across different software artifacts on a semantic level.

In this research, we address important issues for both the Semantic Web and the software maintenance communities. For the Semantic Web community, we illustrate how the use of the semantic technologies can be extended to the software maintenance domain. Furthermore, we demonstrate how the large body of existing knowledge found in source code and software documentation can be made available through automatic ontology population on the Semantic Web.

From a software maintenance perspective, we illustrate through three concrete use cases how the Semantic Web and its underlying technologies can benefit and support maintainers during typical maintenance tasks.

In future versions, more work is needed on enhancing existing software development tools with Semantic Web capabilities, some of which is addressed in the *Semantic Desktop* community. Many of the ideas presented here obviously also apply to other areas in software engineering besides maintenance; we have also been investigating ontology-enabled software comprehension processes [13], which will complement and further enhance the utility of our "Software Semantic Web" approach.

References

1. G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information Retrieval Models for Recovering Traceability Links between Code and Documentation". In Proc. of IEEE International Conference on Software Maintenance, San Jose, CA, 2000.
2. R. Brooks, "Towards a Theory of the Comprehension of Computer Programs". International Journal of Man-Machine Studies, pp. 543-554, 1963.
3. H. Cunningham, D. Maynard, K. Bontcheva, 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. Philadelphia, July 2002.
4. P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard, "LaSSIE - a Knowledge-based Software Information System", Comm. of the ACM, 34(5), pp. 36–49, 1991.
5. V. Haarslev and R. Möller, "RACER System Description", In Proc. of International Joint Conference on Automated Reasoning, Siena, Italy, 2002.
6. P. N. Johnson-Laird, "Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness". Harvard University, Cambridge, MI, 1983.
7. A. V. Mayhauser, A. M. Vans, "Program Comprehension during Software Maintenance and Evolution". IEEE Computer, 28(8), pp. 44-55, August, 1995.
8. IEEE Standard for Software Maintenance, IEEE 1219-1998.
9. D. Jin and J. Cordy. "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology". In Proc. of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, 2005.
10. H.-J. Happel, S. Seedorf, "Applications of Ontologies in Software Engineering", In Proc. of International Workshop on Semantic Web Enabled Software Engineering, 2006.
11. T.C. Lethbridge and A. Nicholas, "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study", Department of Computer Science, University of Ottawa, Technical Report, TR-97-07, 1997.
12. M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" Journal of Systems and Software, 43(1), pp. 19-27, 1998.
13. W. Meng, J. Rilling, Y. Zhang, R. Witte, P. Charland, "An Ontological Software Comprehension Process Model", In Proc. of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering, Italy, 2006.
14. C. Riva, "Reverse Architecting: An Industrial Experience Report", In Proc. of the 7th IEEE Working Conference on Reverse Engineering, Australia, 2000.
15. R. Seacord, D. Plakosh, and G. Lewis, "Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices", Addison-Wesley, 2003.
16. I. Sommerville, "Software Engineering (6th Edition)", Addison-Wesley, 2003.
17. M.A. Storey, S.E. Sim, and K. Wong, "A Collaborative Demonstration of Reverse Engineering tools", ACM SIGAPP Applied Computing Review, Vol. 10(1), pp18-25, 2002.
18. C. Welty, "Augmenting Abstract Syntax Trees for Program Understanding", In Proc. of International Conference on Automated Software Engineering, 1997.

19. R. Witte, T. Kappler, C. Baker, "Ontology Design for Biomedical Text Mining", Chapter 13 in Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences, Springer Verlag, 2006.
20. Y. Zhang, R. Witte, J. Rilling, V. Haarslev, "An Ontology-based Approach for Traceability Recovery", In Proc. of International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering, Genoa, Italy, 2006.
21. A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, C. Welty, "Supporting Online Problem-solving Communities With the Semantic Web", In Proc. of the 15th International Conference on World Wide Web, 2006.
22. H.-J. Happel, A. Korthaus, S. Seedorf, P. Tomczyk, "KontoR: An Ontology-enabled Approach to Software Reuse", In Proc. of the 18th International Conference on Software Engineering and Knowledge Engineering, 2006.