An Ontological Approach for the Semantic Recovery of Traceability Links between Software Artifacts

Yonggang Zhang, René Witte, Juergen Rilling,Volker Haarslev

Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{yongg_zh, rwitte, rilling, haarslev}@cse.concordia.ca

Abstract. Traceability links provide support for software engineers in understanding relations and dependencies among software artifacts created during the software development process. In this research, we focus on re-establishing traceability links between existing source code and documentation to support software maintenance. We present a novel approach that addresses this issue by creating formal ontological representations for both documentation and source code artifacts. Our approach recovers traceability links at the semantic level, utilizing structural and semantic information found in various software artifacts. These linked ontologies are supported by ontology reasoners to allow the inference of implicit relations among these software artifacts.

Keywords: Traceability, Software Maintenance, Ontology, Text Mining

1 Introduction

Software maintenance, often also referred to as software evolution, constitutes a major part of the total cost occurring during the life span of a software system [33, 34]. As software ages, the task of maintaining it becomes more complex and more expensive. Maintainers often face the challenge to identify and comprehend the different representations and interrelationships that exist among software artifacts and knowledge resources involved in software maintenance [35, 37]. Therefore, from a maintainer's perspective, exploring [24] and linking these artifacts and knowledge resources becomes a key challenge [1]. However, it is a well-known fact that even in organizations and projects with mature software development processes, software artifacts created as part of well-defined processes end up to be disconnected from each other [1, 2]. This lack of traceability among software artifacts is caused by several factors, including: (1) the fact that these artifacts are written in different languages (natural language vs. programming language); (2) they describe a software system at various abstraction levels (design vs. implementation); (3) processes applied within an organization do not enforce maintenance of existing traceability links; and (4) a lack of adequate tool support to establish and maintain traceability among various artifacts.

As a result, maintainers tend to spend a large amount of effort on synthesizing and integrating information from various sources in these systems to establish links among artifacts. Existing research in software traceability focuses on reducing the cost associated with this manual effort by developing automatic assistance in establishing and maintaining traceability links among software artifacts [2].

Software documentation, like requirements and design documents, contains a large amount of information in the form of description and text written in natural language. These documents combined with source code represent two of the main software artifact types utilized in software comprehension [1, 2]. Existing source code/document traceability research [1, 26] mainly focuses on connecting documents and source code using Information Retrieval (IR) techniques. However, these IR approaches ignore structural and semantic information that can be found in both documents and source code, limiting therefore both their precision and applicability.

In this research, we introduce a novel formal ontological representation that integrates two of the major software artifacts, source code and software documentation. The goal of using this common representation is to reduce the conceptual gap caused by the type of abstractions and languages found in these artifacts. Having this common representation allows maintainers not only to explore knowledge relevant to their given task across the different modeled artifacts, but also to enrich their current understanding of a system. The use of ontologies for managing traceability between software and other artifacts therefore becomes an important aspect of software language engineering. It provides non-trivial relationships between artifacts and helps with consistency management in a language-based manner.

Unlike existing approaches using IR, we developed a Text Mining system to semantically analyze software documents. The discovered concept instances from both source code and documents are used to establish traceability links between these software artifacts. In addition, the formal ontological representation also allows us to take advantage of automated reasoning services provided by ontology reasoners to infer implicit relations (links) between these two types of artifacts.

Figure 1 here

An overview of our approach is shown in Figure 1. In a first step, ontologies for source code and document artifacts are created and automatically populated from existing source code and documentation artifacts. In a second step,

traceability links among the modeled artifacts are established to allow for querying and reasoning upon this knowledge base to infer implicit relations among the modeled software artifacts.

Our research is significant for several reasons. Firstly, software artifacts other than source code, such as documentation, contain rich semantic information that is not used by existing reverse engineering tools. Introducing an ontological representation for software documentation enables us to utilize Natural Language Processing (NLP) techniques [17] to "understand" parts of the semantics conveyed by these artifacts and to establish additional traceability links among these artifacts.

Secondly, automatic ontology population is an important issue in ontology engineering. In particular for artifacts containing natural language constructs, ontology population with semantically rich information is an ongoing challenge. Our approach demonstrates a feasible solution within the software domain that goes beyond current techniques.

Furthermore, the uniform ontological representation for both source code and documentation allows us to share common concepts between different resources, easing the integration of information by allowing for the automatic recovery and establishment of traceability links among documentation and source code artifacts.

Finally, representing software artifacts in a formal ontology allows programmers to reason about various implicit relations between software artifacts. Taking advantage of these established traceability links and existing ontology-based knowledge representation techniques such as Description Logics [3] and ontology reasoners [9], users can define new concepts and roles (types of relations). This newly modeled knowledge can then be used to support specific maintenance activities (e.g., program comprehension) and consistency checking of artifacts through ontology queries.

The remainder of the paper is organized as follows: in Section 2, we provide relevant background of our research, including formal ontologies and text mining techniques used for ontology population. Section 3 presents our approach to traceability recovery, including a high-level description of our methodology and the requirements that have to be satisfied by the ontology design presented in this section.

Section 4 describes the general system architecture of the Semantic Web-Enabled Software Maintenance Environment we implemented to support our traceability recovery approach. Section 5 covers in detail the automatic population of the software ontology from both source code and documents. Section 6 provides an evaluation and ap-

plication example of our approach. Related work is discussed in Section 7, and conclusions and future work are presented in Section 8.

2. Background

In this section, we introduce background relevant to our research, including ontologies and their formalisms, Description Logics, as well as text mining techniques used for ontology population. Readers familiar with these concepts can safely skip this section.

2.1 Ontology and Description Logics

The term `ontology' originates from philosophy, where it denotes the study of existence. In computer science, the most common definition has been provided by Gruber [8]: “An ontology is an explicit specification of a conceptualization.” Here, ontologies are typically used as a formal and explicit way of specifying the concepts and relationships in a domain of discourse. Description Logics (DL) [3], a family of knowledge representation formalisms, have long been regarded as a standard ontology language. DL is also a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [30]. DL represents the knowledge of a domain by first defining the relevant concepts of the domain in a taxonomy, and then utilizing these concepts to specify properties of individuals occurring in the domain. Furthermore, ontologies allow for the extension of a resulting model to reflect more closely an “open” world assumption [3], by allowing for the integration of incomplete knowledge and extendibility of the ontological model. The use of DL allows for a formal characterization of subsumption relationships between concepts: A concept C is considered a sub-concept of D if all instances of C are also instances of D.

Basic elements of DL are atomic concepts and atomic roles, which correspond to unary predicates and binary predicates in First Order Logic. Complex concepts are then defined by combining basic elements with several concept constructors. For example, in the domain of software design technique and documentation structure, given atomic concepts such as DesignPattern and Paragraph, as well as an atomic role contains that describes a relation between these two concepts, a new concept DesignPatternDoc can be defined by a conjunction constructor and existential qualifier:

$$\text{DesignPatternDoc} \equiv \text{Paragraph} \sqcap \exists\ \text{contains.DesignPattern}$$

Individuals existing in the domain and their relations can be specified as instances of their corresponding concepts and roles. For example, the following DL expressions define *p* as a paragraph instance, *abstract_factory_pattern* as a design pattern instance, and the relation *contains* between the paragraph body *p* and the *abstract_factory_pattern:*

$$\text{p:Paragraph, abstract\_factory\_pattern:DesignPattern, (p, abstract\_factory\_pattern): contains}$$

Having DL as the specification language for a formal ontology enables also the use of reasoning services provided by DL-based knowledge representation systems. Unlike many logic programming approaches that cannot guarantee completeness, DL reasoning services are proven to be sound, complete, and terminating. Moreover, DL reasoning is automatic and does not require the development of logic programs to extract the desired inferences. DL reasoning is usually performed on demand, triggered by relevant queries to the knowledge base. The Racer system [9] is an ontology reasoner that has been highly optimized to support very expressive DLs [3, 9, 15], i.e., DLs that support at least the DL $\mathcal{ALC}$ [3, Chapter 2] extended by transitive roles, inverse roles, role hierarchies, and number restrictions on roles (this logic is denoted as $\mathcal{SHIN}$). An example for an even more expressive DL is $\mathcal{SROIQ}$ [63] that extends $\mathcal{SHIN}$ by qualified number restrictions, nominals, and a restricted form of role composition. Typical 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, tuple retrieval, and instance realization). For example, given the above concept definition of DesignPatternDoc, as well as the assertions about instance *p* and *abstract_factory_pattern*, the ontology reasoner can automatically infer that *p* is also an instance of DesignPatternDoc.

For a more complete coverage of DLs and Racer, we refer the reader to [3, 9].

2.2 Text Mining and Ontology Population

Text Mining is commonly known as a knowledge discovery process that aims to extract non-trivial information or knowledge from unstructured text [11]. Unlike *Information Retrieval* (IR) systems [4], Text mining does not simply return documents pertaining to a query, but rather attempts to obtain *semantic* information from the documents themselves, using techniques from Natural Language Processing (NLP) [17] and Artificial Intelligence. In the software domain, for example, a text mining system can be employed to obtain information about individual software entities mentioned in documents, like the *system architecture*, its *components*, and their *relationships* with *packages* or *classes*. These mentions, called *Named Entities*, can then be exported in a structured format for further (automated) analyses or browsing by a user. Commonly used export formats are XML or relational database tuples. When text mining results are exported in form of instances (individuals) for an existing ontology, this process is also called *ontology population*, which is again different from *ontology learning*, where the concepts themselves (and their relations) are (semi-)automatically acquired from natural language texts.

Text Mining systems are often implemented using component-based frameworks, such as GATE *(General Architecture for Text Engineering)* [5] or IBM's UIMA *(Unstructured Information Management Architecture)*. Within the text mining process, a number of standard NLP techniques are commonly performed. These include first dividing the textual input stream into individual tokens with a (Unicode) *Tokeniser*, using a *Sentence Splitter* to detect sentence boundaries, and running a statistical *Part-of-Speech* (POS) *tagger* that assigns labels (e.g., noun, verb, and adjective) to each word. Larger grammatical structures, such as *Noun Phrases* (NPs) and *Verb Groups* (VGs), can then be created based on these tags using *chunker* modules. Based on these foundational analysis steps, more semantically-oriented analyses can be performed, which typically require domain- and language specific algorithms and resources.

3. Ontology Design for Traceability

We now present our approach to traceability recovery, starting with a high-level description of our methodology. Based on the described approach, we derive a number of requirements (Section 3.2) that have to be satisfied by the ontologies. Our ontology design fulfilling these requirements is then presented in Section 3.3.

3.1 Methodology

As stated above, the goal of our work is the automatic establishment of traceability links between source code and natural language documents. In particular, we are concerned with "deep links" between individual entities on the source code side (e.g., a class or method) and the corresponding mentions of these entities in a document on the level of individual words or phrases. That is, in contrast to Information Retrieval (IR) approaches, we are not interested in creating links on the level of complete documents or paragraphs, but rather much more fine-grained links that precisely show the connections between individual words and code entities. Obviously, these links can only be created between entities that appear on *both* the source code side and documentation side: for example, class names, methods, or variables. But documents may also contain much more higher-level concepts that can not be directly mapped to a single code entity, like descriptions of algorithms, architectures, or requirements. It is here that our ontology-based method provides a strong advantage compared with existing, semantic-poor approaches, as it allows us to explicitly encode knowledge of the software domain in a formal language that can be automatically evaluated, e.g., through queries and reasoning. For example, we can model knowledge about design patterns (e.g., names, types) and their relations with other entities (e.g., classes, methods) in an ontology. An end user, like a software maintainer, can then navigate a single, unified, formal representation that covers both code and documentation, which allows for a number of novel use cases, like the automatic establishment of traceability links, the generation of concept-based document views, and the automatic summarization (slicing) of documents based on an explicit query by a user.

3.1.1 Traceability Link Recovery through Ontology Alignment

Before any more advanced tasks can be performed on our ontological code/document representation, we first have to establish the deep traceability links. This involves a number of steps:

1. Building ontologies modeling the domains of source code and software documents.
2. Creating a knowledge base by automatically populating these ontologies through code analysis and text mining.
3. Establishing traceability links between code and documents through ontology alignment.

Step (1) requires an ontology design that is rich enough to capture the concepts and their relations to support all further analysis steps. These requirements are listed explicitly in Section 3.2 and our resulting design is covered in Section 3.3.

Concepts alone cannot support analysis tasks concerned with a concrete system and its documentation: This requires the creation of a knowledge base containing instances (individuals) for the modeled concepts. The automatic population stipulated by Step (2) is described in Section 4.

Figure 2 here

After populating the ontologies, in Step (3) traceability links are established through ontology alignment [7, 29]. Our approach is illustrated in Figure 2. The upper part shows concepts of the document and source code domain. Instances of these concepts are detected through the analysis of concrete systems, shown in the bottom half. The traceability links between code and documentation are created through ontology alignment, based on class and instance information. Since our documentation ontology and source code ontology share many concepts from the programming language domain, such as Class or Method, we focus in this research on matching instances, based on their names and properties.

The result is a single formal, ontology language-based representation of both source code and natural language that can now be utilized by numerous higher-level tasks.

3.1.2 Querying code and documentation in a single representation

The linked source code and documentation ontologies enable us to combine knowledge from both software implementation and documentation. Using the established links between the source code and documentation ontology, a user can perform ontological queries on either documents or source code using any of the contained concepts, entities, or relations. Note that this is a much more powerful paradigm than IR-based analysis [1, 26], which only returns a (ranked) list of documents without further structure.

Questions that might be asked by a software maintainer, and encoded into an ontology query, are:

- What are the document passages (sentences/paragraphs) describing the source code entity (class, method, etc.) I am currently visiting in my editor?
- What design patterns are mentioned in the code's documentation within the same paragraph as the class I am currently working on?
- What architectural descriptions are contained in the documentation? What are its parts, and how do they relate to the source code base?

Note that these queries cross the conceptual boundary between code and natural language documents, which is not possible with other approaches that lack the ability to model both kinds of artifacts in the same formal representation. In fact, queries can also be asked from the viewpoint of a documentation writer, asking for all code segments related to a text currently being written or edited.

Query Example. As an example, assume that the populated document ontology contains the class *AST* as part of a Visitor pattern. 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:*

```
var query = new Query();                                              // define a new query
query.declare("P", "C");                                              // declare two query variables
query.restrict("P", "Paragraph");                                     // P is a paragraph
query.restrict("C", "Class");                                         // C is a class
query.restrict("C", "hasSuper", "net.refractions.udig.catalog.util.AST"); // C is a sub-class of AST
query.restrict("P", "contains", "C");                                 // P contains C
query.retrieve("P");                                                  // this query only retrieve P
var result = ontology.query(query);                                   // perform the query
```

This query utilizes both the programming language semantics, such as the inheritance relation between query variable *C* and the class *AST* (direct and indirect), and the structural information of documentation, such as the contains relation between *P* and *C*. The result of this query therefore contains all text paragraphs that describe the sub-classes of *AST*, i.e., the Visitor pattern [12]. 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.

3.1.3 Documentation slicing and focused summarization

The potentially large amount of documentation associated with a legacy system is the motivation for delivering more advanced, semantically-oriented navigation features to software development environments. Often, information pertaining to a certain entity, like a class, is distributed over many different documents, which makes it difficult for a developer to quickly gain a comprehensive overview on all the available knowledge. With our ontology model, we can deliver a *virtual* document containing all the information pertaining to a specific entity (or a set of entities) of

interest to a software engineer. This is also known as *(focused) automatic summarization* in the area of NLP and can be compared with *slicing* of documents, instead of source code.

Documents can not only be re-structured based on their links to source code, but also using the design-level concepts that relate to particular maintenance tasks. For example, 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 3).

Figure 3 here

Document Slicing Example. For example, a Visitor Pattern [12] within a document can be considered as all the text paragraphs that describe or contain information related to the concept Visitor. The following new concept VisitorPattern-Doc can be used to retrieve paragraphs that relate to such a Visitor Pattern. Similarly, a new concept HighlevelDoc can be also defined to retrieve all documents that contain the high-level design concept Architecture or DesignPattern. The ontology reasoner can then automatically classify documents according to these concept definitions.

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

$$\text{HighlevelDoc} \equiv \text{DocumentFile} \sqcap \exists\ \text{contains.(Architecture} \sqcup \text{DesignPattern)}$$

3.2 Detected Requirements

Based on the stated goal and approach, we can now define the requirements for traceability recovery systems using the ontology language OWL-DL more precisely.

Requirement #1: Ontology Design for Traceability. Our approach relies on the existence of a single formal language model for both source code and documentation. Both types of artifacts must be modeled in enough detail to

allow the establishment of the "deep links" stipulated above. In particular, the documentation and source code sub-hierarchies must be modeled such that they share a number of concepts: otherwise, cross-artifact queries and reasoning as exemplified above would not be possible. This requirement is addressed in Section 3.3.

Requirement #2: Ontology Implementation. To illustrate the feasibility of this approach, it is necessary to implement the ontologies and create a program comprehension environment that manages the various software entities and the ontologies. As stated in the background section, this implementation should be based on established standards, such as the W3C standard OWL. We discuss implementation issues in Section 4. Furthermore, the implementation must be based on an ontology language that supports queries and formal reasoning, which are required for the more advanced traceability use cases, such as document summarization.

Requirement #3: Automated Ontology Population. Creating a knowledge base, i.e., populating the ontology with instances, is a necessary prerequisite for an automated environment. Thus, automatic population methods must be developed that can take existing source code and its accompanying documentation and create instances in the modeled ontology. Our ontology population subsystems are described in detail in Section 5.

Requirement #4: Automatic Traceability Recovery. After populating the software ontology, traceability links must be identified and created, based on the ontology alignment strategy outlined above.

In addition to these fundamental requirements, this work also requires a formal evaluation, based on clearly defined tasks, metrics, and data, as well as a demonstration of its usefulness within a real-world scenario. We address these points in our evaluation and application section (Section 6).

3.3 Ontological Representation of Software Artifacts

We start addressing the stated Requirement #1 by designing our *software ontology*. The goal of our design is to cover the rich structural and semantical knowledge contained in software artifacts, in particular for source code and docu-

mentation. This underlying ontological model is one of the major differences between our semantic approach and common “bag-of-words” approaches used in Information Retrieval [1, 26].

To satisfy our second requirement, we employ the formal ontology standard OWL-DL. These *Web ontologies* are an emerging technology for capturing the semantics in a domain of discourse. Based on available *Semantic Web* technologies, they also allow us to deliver further semantic support to end users [40] that goes beyond traceability recovery.

Our software ontology contains two major sub-ontologies for representing *source code* and *documents* within the software domain, described in detail in the following two subsections.

3.3.1 Source Code Ontology

The source code sub-ontology contains more than 100 common concepts from the programming language domain, as well as some additional concepts that are *Java* specific. Figure 4 shows a partial view of the Java taxonomy modeled in the source code ontology.

Figure 4 here

As suggested in [31], concepts in a domain can be classified into two categories, objects and actions. Objects typically refer to entities that may appear in the domain, and actions refer to their associations.

The SourceObject concept is introduced to denote all entities that may appear in the source code. Sub-concepts of SourceObject include SourceFile, Package, Class, Method, Variable, Statement, and Expression, etc. Each of these concepts typically corresponds directly to some programming language constructs in the source code, and therefore, instances of these concepts can be automatically identified by source code parsing. This is discussed in detail in Section 5.1.

A SourceAction typically denotes a semantic association between two or more SourceObjects. For example, a MethodCall action refers to a method invocation expression within the body of a method that invokes another method. The properties of MethodCall therefore include the caller method, the called method, the types of parameters sent by this method call, and the return type of the called method.

The object part of the ontology (concept SourceObject and its sub-concepts) is designed to contain complete syntactical information of the source code. The action part (SourceAction and its sub-concepts) consists of selected semantic information that is of particular interest to software maintainers.

Several utility concepts are defined to specify properties of other source code entities. For example, the concept Type represents an abstract type of source code entities and a Modifier concept is used to specify the accessibilities of entities, such as public, private, etc.

The use of DL allows the formal characterization of subsumption relationships between concepts. A concept C is considered as a sub-concept of D if all instances of C are also instances of D. As a result, if an individual is specified as a Method in our ontology, it will be automatically recognized as a Member, and further, as a SourceObject.

In addition, we provide *necessary conditions* for all the concepts in the ontology using different concept constructors. For example,

$$\text{Method} \sqsubseteq \text{SourceObject} \sqcap \exists \text{definedIn.Class}$$

describes a situation in which a method is a type of source object, and it has to be defined in a class, and

$$\text{Variable} \sqsubseteq \forall \text{hasType.Type}$$

means that a variable must have a type. Some concepts are defined by providing *necessary and sufficient conditions*. For example,

$$\text{Member} \equiv \text{Method} \sqcup \text{Field}$$

can be interpreted as a class member that is either a method or a field, and

$$\text{PublicField} \equiv \text{Field} \sqcap \exists \text{ hasModifier.PublicModifier}$$

defines that a public file is a field with public modifiers.

Relationships. Within the source code ontology, many roles are defined to specify relationships among concepts. A partial view of the role names and their descriptions found in our source code ontology is shown in Table 1.

Table 1 – Role Names in Source Code Ontology

Role Name	Description
definedIn	SourceObject A is defined in SourceObject B
hasSuper	Class A has super Class B
hasSub	Class A has sub Class B
Call	Method A calls Method B
indirectCall	Transitive relation of method call
Access	Method A read/write Variable B
Read	Method A read Variable B
Write	Method A write Variable B
hasType	Variable A has Type B
typeOf	Variable A is of Type B
Create	Class A creates instance of Class B

Additionally, we also provide for each of these roles their corresponding *inverse role*. For example, the inverse role of hasSuper is hasSub, i.e., if class C_1 hasSuper class C_2, then C_2 also hasSub class C_1. Similarly, if a method M write a variable V, then V is writtenBy M.

One of the advantages of using DLs is its ability to define *transitive roles*. If a role R is defined as a transitive role, and if $(a,b)\in R$ and $(b,c)\in R$ are specified, then $(a,c)\in R$ also holds. Transitive roles are especially useful for specifying part-of relationships between source code elements (through definedIn role), inheritance relationships between classes (through hasSuper role), and indirect calling relationships (through indirectCall role).

The definition of *subsumption* relationships between roles is also supported. More formally, if role R is a sub-role of S and (a, b)∈R holds, then (a, b)∈S will also hold. For example, in our ontology, the read role is a sub-role of access. Let *M* and *V* be instances of Method and Variable respectively, if a relation (M, V)∈read is discovered, then (M, V)∈access is also implied.

More expressive DLs allow defining a role as the *composition* of two or more roles. If both (a, b)∈R and (b, c)∈S hold, and a role T is defined as T ≡ R ○ S, then (a, c)∈T also holds. For example, in our ontology, a role classReadVariable can be defined as the composition of contain and read:

$$\text{classReadVariable} \equiv \text{contain} \circ \text{read}$$

In such a case, if class C contains a method M, and the method M reads a variable V, that is, (C, M)∈contain, and (M, V)∈read, then (C, V)∈classReadVariable is also implied.

Instances of roles (i.e., relationships between source code entities) are often implicit, but can still be recognized through static source code analysis. For example, if within the body of a method, a variable is found in the left hand side (LHS) of an assignment expression, such as –

```
public int aMethod(){
        ……
        aField = 1;
        ……
}
```

then an instance of the write role will be created, indicating that the method writes the variable, like: (aMethod, aField) ∈ write.

Characterizing Object-Oriented Programming. We designed our ontology to support three key features specific to the Object-Oriented Programming (OOP) paradigm, namely encapsulation, inheritance, and polymorphism.

(1) *Encapsulation:* A hasModifier role is defined to specify whether a SourceObject (e.g., Class, Method, or Field) is public, private, or protected.

(2) *Inheritance:* We define hasSuper and hasSub to represent class inheritance. The hasSuper and hasSub roles are transitive.

(3) *Polymorphism:* As described earlier, we use static source code analysis techniques to populate the source code ontology. Polymorphism is often used to specify a particular dynamic aspect of OOP languages, which can only be captured partially by static analysis approaches. Within our ontology, we provide an override role to denote that a method overrides another method defined in the superclass. A sameSignature role is also defined to represent the relationship between two methods that are defined in different subclasses and share the same signature, i.e., they both override the same method defined in the superclass. If two methods are in a sameSignature relationship, both of them may potentially be invoked when the method in the superclass is invoked, e.g., through dynamic binding (polymorphism).

Figure 5 here

For example, as shown in the UML class diagram in Figure 5, both SubClassA and SubClassB inherit from BaseClass. Therefore, the following relations can be automatically identified:

$$(\text{SubClassA.baseMethod}, \text{BaseClass.baseMethod}) \in \text{override}$$

$$(\text{SubClassB.baseMethod}, \text{BaseClass.baseMethod}) \in \text{override}$$

and

$$(\text{SubClassA.baseMethod}, \text{SubClassB.baseMethod}) \in \text{sameSignature}$$

Both the override and the sameSignature role are transitive. Override is a sub-role of the symmetric sameSignature role.

3.3.2 Documentation Ontology

The documentation ontology further extends the source code ontology with a large body of concepts that can be discovered in software documents. At the current stage of our research, this ontology is mainly based on concepts found in the software domain, including concepts such as programming languages, algorithms, data structures, and design decisions, especially design patterns and software architectures. Figure 6 shows the top-level concepts of the documentation ontology.

Figure 6

The software documentation ontology has been designed to support automatic ontology population through a text mining system by adapting the ontology design requirements outlined in [41] 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 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.

Relationships between the classes, including the ones modeled in the source code ontology. These relationships allow us to automatically restrict NLP-detected relations to semantically valid ones (e.g., a relationship 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 can also be utilized for detecting corresponding entities in documents [39].

4. System Architecture and Implementation

In this section, we describe the general system architecture of our *Semantic Web-Enabled Software Maintenance Environment*, and illustrate how the detected Requirement #2 of such a system are fulfilled by our implementation. In particular, we describe the overall architecture of our approach in Section 4.1 and some implementation details in Section 4.2. A particular feature of our implementation, an Eclipse plug-in for querying the ontology, is further described in Section 4.3.

4.1 Architecture Overview

Our system is based on common Semantic Web infrastructures, including an ontology reasoner, a knowledge management system, and public RDF/OWL APIs. In order to utilize the structural and semantic information found in various software artifacts, we have developed source code analysis and text mining sub-systems that can automatically extract concept instances and their relations from source code and documents, respectively. An *Eclipse* integrated user interface is provided to support the querying and exploring of the populated software ontology. Our system has been designed to enable software maintainers in both discovering concepts and relations within a software system, as well as automatically inferring implicit relations among different artifacts. The automatic population of the software ontology stipulated by Requirement #3 is handled by two sub-systems for source code analysis and text mining, described in more detail in the next section. The discovered instances can then be linked through ontology alignment [7, 29] techniques. Based on the software ontology, users can also define new concepts or instances for particular reverse engineering tasks through an ontology management interface.

Figure 7 here

4.2 System Implementation

The right side of Figure 7 shows the software ontology as described previously in Section 3.3. The two sub-ontologies are modeled in OWL-DL [30, 46] and were created using the Protégé-OWL extension of Protégé [47], a free ontology editor.

Racer [9], 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 maintenance domain, where one has to process a large amount of instances efficiently.

Automatic ontology population is handled by two subsystems: The source code analysis and the document analysis. The source code analysis subsystem is based on JDT [48], which is a Java parser provided by Eclipse [49].

The system analyzes the Abstract Syntax Tree (AST) of Java source code and identifies entities and their relations to populate the source code ontology.

The text mining subsystem has been implemented based on the GATE *(General Architecture for Text Engineering)* framework [5], one of the most widely used NLP tools. Starting with version 3.0, GATE has been featuring built-in ontology support in form of an abstraction layer between the components of an NLP system and the various ontology representations. This layer is built on Jena [50] as RDF-Store, which enables the use of OWL ontologies from within GATE.

4.3 Query Interface

Within our system, a software engineer can use the Racer query language nRQL [15] to retrieve instances of concepts and roles in the ontology. An nRQL query uses arbitrary concept names and role names in the ontology to specify properties of the result. In a query, variables can be used to bind to instances that satisfy the query.

However, the use of nRQL queries is still largely restricted to users with a high mathematical/logical background due to nRQL's syntax, which, although comparatively straightforward, is still difficult for programmers to understand and therefore may be difficult to apply. To bridge this gap between practitioners and Racer, we have introduced an additional scripting language, based on *JavaScript*, as a query language. We introduced a set of built-in functions and classes in the JavaScript interpreter *Rhino* [51] to simplify querying the ontology for users.

Within the JavaScript interpreter, we provide a set of logic functions for formulating complex concepts. Using these logic functions, users can construct their own concepts. For example, the concept ClassMember discussed in Section 3.3.1 can be specified using the built-in functions as:

ontology.define_concept("DesignPatternDoc", AND("Paragraph, EXIST("contains", "DesignPattern")))

Two classes, *Query* and *Result*, are provided to assist users in composing queries and manipulating the results. Users can arbitrarily use the vocabulary in the ontology to retrieve instances with specified properties. The typical procedure of composing a query is as follows: (1) query variables are declared; (2) restrictions that apply to the variables are specified using concepts, roles, and instances in the ontology; and (3) the query is submitted to the built-in JavaScript object called "ontology".

The result of a query is a set of tuples that satisfy the specified restrictions. For example, the following query script retrieves all paragraphs that contain design pattern instances from the documentation ontology:

```
var design_pattern_doc = new Query();                    // create an new query
design_pattern_doc.declare("P", "DP");                   // declare two query variables in the query
design_pattern_doc.restrict("P", "Paragraph");           // restrict P to be bound to a paragraph instance
design_pattern_doc.restrict("DP", "DesignPattern");      // restrict DP to be bound to a design pattern instance
design_pattern_doc.restrict("P", "contains", "DP");      // restrict P contains DP
design_pattern_doc.retrieve("P");                        // the query will only retrive instances of P
var result = ontology.query(design_pattern_doc) ;        // perform the query
```

The query first declares two variables P and DP, and then specifies that P shall be bound to an instance of Paragraph, and DP to an instance of DesignPattern. The third restriction specifies that P and DP shall have a contains relation. The next statement states that this query only retrieves instances bound to P.

The scriptable query language allows users to benefit from both the declarative semantics of Description Logics as well as the fine-grained control abilities of procedural languages.

The query interface of our system is a plug-in that provides OWL integration for the Eclipse software development platform. Table 2 compares our JavaScript query language with the nRQL and SPARQL [52] languages through a simple query.

Table 2. Query Comparison - Java Script Interface, nRQL and SPARQL

JavaScript	nRQL	SPARQL
var query = new Query(); *query.declare("M", "V", "C");* *query.restrict("M", "definedIn", "C");* *query.restrict("M", "contains", "V");* *query.retrieve("C", "V");* *var result = ontology.query(query);*	*(RETRIEVE (?C ?V)* *(AND* *(?M ?C definedIn)* *(?M ?V contains)))*	*PREFIX sc: <http://...>* *SELECT ?C ?V* *WHERE* *{ ?M sc:definedIn ?C .* *?M sc:contains ?V }*

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 (Requirement #3). 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 Source Code Ontology Population

Concepts in the source code ontology typically have a direct mapping to source code entities, allowing instances of these concepts to be automatically recognized by our source code ontology population subsystem. Within our implementation, we utilize the Eclipse JDT compiler in order to read the source code, performing 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 8). Furthermore, it can also identify instances of roles (i.e., relations between source code entities) by statically analyzing the source code.

Figure 8 here

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 Figure 8. 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. 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, we limit the population of the source code ontology to 38 of the higher-level concepts (classes) and 41 types of relations (ObjectProperties). We restrict the ontology population currently to these high-level concepts (e.g. package, class, method) due to the application domain of the ontology. Our objective is currently to support a more general system comprehension rather than focusing on specific, low-level source analysis (e.g., at the expression level).

We have performed several case studies on different open source systems to evaluate the size of the populated source code ontology. The results are described in Section 6.

5.2 Documentation Ontology Population

We developed a custom text mining system to extract knowledge from software documents and populate the corresponding sub-ontology [39]. 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.

The system first performs a number of standard preprocessing steps, such as tokenisation, sentence splitting, part-of-speech tagging, and noun phrase chunking [53]. Then, named entities 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.

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 occurrences, we perform an additional co-reference resolution step to detect both nominal and pronominal 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.

For further details on our software text mining system, we refer the reader to [39].

6. Evaluation and Application

In this section, we analyze the performance of our approach. The discussion is split into two parts: In the first subsection, we present quantitative numbers for the automatic ontology population. The second subsection then demonstrates the applicability of our approach on a real-world code base.

6.1 Ontology Population Evaluation

In what follows, we first analyze the performance of our system, in particular with respect to the automatic population of the source code and document ontologies.

Source Code Analysis Performance. Table 3 summarizes the evaluation results of the automatic source code population subsystem. The information provided in the table includes the size of the software system being measured in lines of code (LOC). The processing time includes both AST traversal and ontology population time in seconds. The size of the resulting source code ontology is measured in terms of concept instances and relations identified as shown in the last two columns of the table. As it can be observed from Table 3, for the small to medium size systems the processing time, number of instances and relations are directly dependent on the size of the system. It has to be noted that for the analysis of (very) large systems on computers without sufficient memory available to store the complete AST and ontology in memory, the processing time might increase significantly.

Table 3: Source code and generated Ontology sizes

	LOC	Proc. Time	Instances	Relations
java.util	24k	13.62s	10140	47009
InfoGlue [54]	40k	27.61s	15942	77417
Debrief [55]	140k	67.12s	52406	244403
UDig [56]	177k	82.26s	69627	284692

Text Mining Performance. So far [39], we evaluated our text mining subsystem on two collections of texts: a set of 5 documents (7743 words) taken from the Java documentation for the *Collections* framework [57] and a set of 7 documents (3656 words) from the documentation of the uDig [58] 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 forms, as well as relations between the entities. In what follows, we present results for the named entity recognition and entity normalization. For performance evaluation of the software named entity recognition, we computed the standard *precision, recall,* and *F-measure* results. A named entity was only counted as correct if it matched both the textual description and ontology class. Table 4 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 named entity detection as described in [39].

Table 4. Text Mining evaluation results: Entity recognition and normalization performance

Corpus	**Text Mining Only**				**With Source Code Ontology**			
	P	*R*	*F*	*A*	*P*	*R*	*F*	*A*
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%

As it can be observed, the text mining system achieves a very high precision (90%) in the named entity detection task, with a recall of 62%. With the imported source code instances, 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 strategies when importing entities from the source code analysis, which are currently under investigation. However, the current results already underline the feasibility of our approach of integrating code analysis and NLP.

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 4 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 (column A), the normalization component's performance is very satisfying.

6.2 Application Evaluation

An initial evaluation of the practical applicability of our approach has been performed on a large open source Geographic Information System (GIS), the "User-friendly Desktop Internet GIS" (uDig) [59]. The uDig system is implemented as a set of Eclipse plug-ins that provide geographic information management integration. The uDig documents used in the study consist of a set of JavaDoc files and a requirements analysis document [60].

Links between the uDig implementation and its documentation are recovered by first performing source code analysis to populate the source code ontology (cf. Section 5.1). The resulted ontology contains instances of Class, Method, Field, etc., and their relations such as inheritance, invocation, etc. Our text mining system (cf. Section 5.2) then takes the identified class names, method names, and field names as input to populate the documentation ontology. 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 [12] or architectural styles [32]. The ontology alignment rules are then applied to link both the documentation ontology and the source code ontology. Parts of our results are shown in Figure 9, with the content of the figure corresponding to the following sentences:

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 9 shows that in the uDig documents, our text mining system was able to discover that a sentence (*sentence_2544*) contains both 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 *net.refractions.udig.catalog.IGeoResource*, respectively. In addition, in another sentence *(sentence_712)*, a class in-

stance (*_719_AST*) and a design pattern instance (*_718_visitor_pattern*) are also identified. Instance *_719_AST* can then be linked in a similar manner to the *net.refractions.udig.catalog.util.AST* interface in the source code ontology.

Figure 9 here

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 maintainer may want to study the classes that are related to *FeatureStore*. Within the source code ontology, a query similar to the following script (Script 1) can be performed to retrieve all classes that contain methods that call class *FeatureStore*.

```
var query = new Query();                                          // define a new query
query.declare("M1", "M2",  "C");                                  // declare three query variables
query.restrict("M1", "Method");                                   // M1 is a method
query.restrict("M2", "Method");                                   // M2 is also a method
query.restrict("C", "Class");                                     // C is a class
query.restrict("M1", "definedIn", "C");                           // M1 is defined in C
query.restrict("M2", "definedIn", "org.geotools.data.FeatureStore"); // M2 is defined in FeatureStore
query.restrict("M1", "calls", "M2");                              // M1 calls M2
query.retrieve("C");                                              // this query only retrieve C
var result = ontology.query(query);                               // perform the query
```

Script1 – Query on Source Code Ontology

Unfortunately, the class *IGeoResource*, which has a documented relation with *FeatureStore* (Script 1), will not be returned by such a query, because *IGeoResource* has no explicit invocation relations with *FeatureStore* in the uDig implementation. 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 types of queries are supported due to the already established links between the source code and documentation ontology. For example, the following query (Script 2)

retrieves all classes that occur in the same sentences as class *FeatureStore*. This time, class *IGeoResource* will be returned because both classes occur in *sentence_2544*. The retrieved classes, as well as the associated sentences therefore provide additional information useful for reverse engineers to understand the class *FeatureStore*.

```
var query = new Query();                                          // define a new query
query.declare("S", "C");                                          // declare two query variables
query.restrict("S", "Sentence");                                  // S is a Sentence
query.restrict("C", "Class");                                     // C is a Class
query.restrict("S", "contains", "org.geotools.data.FeatureStore"); // S contains FeatureStore
query.restrict("S", "contains", "C");                             // S also contains C
query.retrieve("C", "S");                                         // retrieve C and the sentence S
var result = ontology.query(query);                               // perform the query
```

Script 2 – Query on Documentation Ontology

The linked source code and documentation ontologies also provide the ability 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 9). In order to retrieve all documented information related to the detected pattern, the following query (Script 3) can be used to retrieve all text paragraphs that describe the sub classes of *AST*.

```
var query = new Query();                                          // define a new query
query.declare("P", "C");                                          // declare two query variables
query.restrict("P", "Paragraph");                                 // P is a paragraph
query.restrict("C", "Class");                                     // C is a class
query.restrict("C", "hasSuper", "net.refractions.udig.catalog.util.AST"); // C is a sub-class of AST
query.restrict("P", "contains", "C");                             // P contains C
query.retrieve("P");                                              // this query only retrieve P
var result = ontology.query(query);                               // perform the query
```

Script 3 – Query Across the Source Code and Documentation Ontology

This query utilizes both, the programming language semantics, such as the inheritance relation between query variable *C* and the class *AST*, and the structural information of documentation, such as the containing relation between *P* and *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.

7. Related Work and Discussions

There exists a significant body of work on analyzing and modeling source code and its semantic structure [13, 14], however these approaches are mostly limited to modeling and analyzing source code artifacts and do not include the recovery of traceability between source code and documents written in natural language.

Traditional approaches dealing with software documents are mainly based on Information Retrieval (IR) techniques [4], which address the indexing, classifying, and retrieving of information in natural language documents. Some existing research recovers traceability links between source code and design documents using IR techniques by indexing software documents and then automatically linking software requirements documents through these indexes to implementation artifacts. Antoniol et al. [1] addressed the traceability problems by linking software documents and code using both probabilistic methods and a Bayesian classifier. They apply their information retrieval model on two case studies to trace C++ source code onto manual pages and Java code to functional requirements. Marcus et al. [25, 26] used Latent Semantic Indexing (LSI) [23], which is a machine-leaning model that induces representations of the meaning of words by analyzing the relationships between words and passages in large bodies of text, to recover traceability links and to locate domain concepts in source code. Their semantic indexing approach extracts the meaning (semantics) of the documentation and source code, and then uses this information to identify traceability links based on similarity measures. Hayes et al. [10] also demonstrated a tool that uses a vector space model to improve requirements tracing.

In contrast to these IR approaches, our work also utilizes structural and semantic information found in both the documentation and the source code by means of text mining and source code parsing. This additional information allows us to recover links that would otherwise not be discovered using traditional IR techniques [4, 11]. Classical IR models only compare the occurrences of individual keywords. However, in many situations, synonyms and concept hierarchies (general concept vs. specific concept) have to be taken into account. For example, the query "people" should also match the document that contains "person", "student", or "professor", as "person" is a synonym of "people", and "student" and "professor" are both more specialized concepts of "people". Thesaurus-based retrieval model [22] and ontology-based models [18] address this issue by utilizing a collection of information concerning relationship between different terms. However, IR based approaches typically neglect structural and semantic information in the software documents, therefore limiting their ability in providing results with regards to the "meaning" of documents.

Very little previous work exists on text mining software documents containing natural language. 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 [16, 27] or detect inconsistent requirements [21]. In contrast, we aim to support the automatic recovery of traceability links for legacy systems. 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 identified through source code analysis, which is an important contribution of our work. Furthermore, compared to logic programming approaches, using ontologies enables us to extend our resulting model to reflect more closely an "open" world assumption [3], allowing for the integration of incomplete knowledge (artifacts) and extendibility of our ontological knowledge base to reflect more closely the creation of mental models used during program comprehension [35].

Ontologies have been commonly regarded as a standard technique for representing domain semantics and resolving semantic ambiguities. Existing research on applying Description Logics or formal ontology in software engineering have been addressed in early works of the LaSSIE [6] and CBMS [35] systems. In addition to above knowledge-based software engineering tools, other researchers have proposed to apply DL in the software engineering domain. In [36], Welty et al. presented a formal ontology that may capture and re-use architectural level knowledge from

software documents. Möller [28] presented an approach that integrates OO programming methodologies and DL knowledge systems. In [43], Yang et al. presented an approach that can extract domain ontologies from legacy systems written in COBOL for program comprehension and re-engineering. Ontologies have also been used in software engineering to establish a common terminology or to model specific aspects of software engineering processes [42, 38]. Compared with our approach, these systems are however much more restricted by the expressiveness of their underlying ontology languages. In addition, these systems also lack the support of optimized DL reasoners, such as Racer in our case.

The introduction of an ontological representation for software artifacts allow us to utilize existing techniques, such as Text Mining and Information Extraction [11], to "understand" parts of the semantics conveyed by these informal information resources, and thus to integrate information from different sources at finer granularity levels. Note that our approach relies on an ontology that has been explicitly designed for text mining, unlike existing approaches employing text mining for ontology learning, like the work by Sabou [61, 62]. We believe that the current methods for automated ontology construction do not provide the necessary level of detail as discussed in Section 3.3. However, a future combination of both approaches, i.e., automatically enriching a pre-designed ontology with newly detected concepts, might deliver additional benefits, for example, in dealing with domain-specific information.

In our previous work, we have already demonstrated how the ontological model of source code and documentation can support various maintenance tasks, such as program comprehension [38], architectural analysis [40], and security analysis [44]. In another work, we have examined the requirements for software reverse engineering repositories [20], where we focus on dealing with incomplete and inconsistent knowledge on software artifacts obtained from different sources. We aim at incorporating this work into our ontology approach to systematically deal with the possible inconsistencies between source code and document analysis discovered during the automatic ontology population.

8. Conclusions and Future Work

The presented research addresses an important issue in the reverse engineering domain, the recovery and maintenance of traceability links among existing documents and source code artifacts. We present a novel approach that provides formal ontological representations for both source code and document artifacts. The ontologies capture structural and

semantic information conveyed in these artifacts, and therefore allow us to recover traceability links between the software implementation and documentation at the semantic level.

In addition, utilizing state-of-the-art ontology reasoners such as Racer, our approach also allows inferring implicit relations between discovered concept instances. The linked ontologies provide the capability to perform queries across the boundary between programming language and natural language.

Furthermore, our documentation ontology identifies different design-level concept instances such as design patterns and architectural styles. These identified instances, linked to source code entities, allow users to discover relations between source code and its corresponding design information at different levels of abstraction.

As part of our future work, we will be exploring a hierarchical linking strategy, starting from code, including inline comments (like JavaDoc), over implementation, design, and specification documents to domain-specific knowledge, to allow us to offer a truly complete process life-cycle for the automated support of traceability links.

ACKNOWLEDGEMENTS

Qiangqiang Li contributed to the software text mining system.

REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, “Information retrieval models for recovering traceability links between code and documentation”. In Proceedings of IEEE International Conference on Software Maintenance, San Jose, CA, 2000

[2] P. Arkley, P. Mason, and S. Riddle, “Position Paper: Enabling Traceability,” Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland (September 2002), pp. 61–65.

[3] F. Baader et al., “The Description Logic Handbook”, Cambridge Univ. Press, 2nded, 2007.

[4] R. Baeza-Yates, & B. Ribeiro-Neto, (1999). Modern Information Retrieval. Addison Wesley.

[5] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan. “GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications.” Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia, July 2002.

[6] P.Devanbu, R.J.Brachman, P.G.Selfridge, and B.W.Ballard, “LaSSIE: a Knowledge-based Software Information System”, Com. of the ACM, 34(5):36–49, 1991.

[7] M. Ehrig, `Ontology Alignment: Bridging the Semantic Gap'. Springer, 2006.

[8] T. Gruber, `A Translation Approach to Portable Ontology Specifications'. Knowledge Acquisition, Vol.5, No.2, 1993, Academic Press, pp.199-220.

[9] V. Haarslev and R. Möller, “RACER System Description”, In Proc. of International Joint Conference on Automated Rea-

soning, IJCAR'2001, Italy, Springer-Verlag, pp. 701-705.

[10] J. H. Hayes, A. Dekhtyar, & Osborne, J. (2003). Improving Requirements Tracing via Information Retrieval, Proceedings of the 11th IEEE International Requirements Engi-neering Conference, pp. 138-147.

[11] R. Feldman, & J. Sanger, (2006). The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data, Cambridge University Press

[12] E.Gamma, R.Helm, R.Johnson, and J.Vlissides "Design Patterns – Elements of Reusable Object-Oriented Software", Addi-son-Wesley, 1994

[13] Y. Gueheneuc, K. Mens R. Wuyts, A comparative framework for design recovery tools, Proceedings of the Conference on Software Maintenance and Reengineering, Bari, Italy, March 22

[14] Y. Guéhéneuc, Ptidej: Promoting Patterns with Patterns. Proceedings of the first ECOOP workshop on Building a System using Patterns, July 2005. Springer

[15] V. Haarslev, R. Möller, and M. Wessel, "Querying the Semantic Web with Racer + nRQL", In Proc. of the KI-2004 Inter-national Workshop on Applications of Description Logics (ADL'04), Ulm, Germany, September 24, 2004.

[16] M.G. Ilieva and O. Ormandjieva. "Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation". 10th Intl. Conf. on Applications of Natural Language to Information Systems (NLDB), Alicante, Spain, 2005.

[17] D. Jurafsky and J. H. Martin, "Speech and Language Processing**"**. Prentice Hall, 2000.

[18] L. Khan, D. McLeod, & E. Hovy, (2004). Retrieval Effectiveness of an Ontology-based Model for Information Selection. International Journal on Very Large Data Bases, 13(1), pp. 71-85.

[19] A. Kiryakov, B. Popov, I. Terziev, D. Manov, and D. Ognyanoffe "Semantic Annotation,Indexing, and Retrieval". Journal of Web Semantics, vol. 2(1), 2005.

[20] U. Kölsch and R. Witte, "Fuzzy Extensions for Reverse Engineering Repository Models". 10th Working Conference on Reverse Engineering (WCRE), Canada, 2003.

[21] L. Kof. "Natural Language Processing: Mature Enough for Requirements Documents Analysis?" 10th Intl. Conf. on Appli-cations of Natural Language to Information Systems (NLDB), Alicante, Spain, June 15-17, 2005.

[22] G. Kowalski (1997). Information Retrieval Systems: Theory and Implementation, Kluwer Academic Publishers.

[23] T. K. Landauer, P.W. Foltz, & D. Laham (1998). An Introduction to Latent Semantic Analysis. Discourse Processes, 25, pp. 259-284.

[24] 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.

[25] A. Marcus, A. Sergeyev, V. Rajlich, & J. I. Maletic (2004). An Information Retrieval Approach to Concept Location in Source Code, 11th IEEE Working Conference on Reverse Engineering. Netherlands.

[26] A. Marcus, J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing". In Proceedings of 25th International Conference on Software Engineering, 2002

[27] V. Mencl. "Deriving Behavior Specifications from Textual Use Cases". Proc. of Workshop on Intelligent Technologies for Software Engineering (WITSE'04), Austria, 2004.

[28] R. Möller (1996). A Functional Layer for Description Logics: Knowledge Representa-tion Meets Object-Oriented Pro-gramming, OOPSLA´96. San Jose, California.

[29] N. F. Noy and H. Stuckenschmidt, "Ontology Alignment: An annotated Bibliography – Semantic Interoperability and Inte-gration" Dagstuhl, Germany, 2005

[30] OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, URL: http://www.w3.org/TR/owl-ref/

[31] B.Schneiderman, "An Empirical Studies of Programmers". In Proceedings of 2nd Workshop on Empirical Studies of Pro-grammers. Ablex Publichers, NJ, 1986

[32] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall Publisher, 1996.

[33] R. Seacord, D. Plakosh & G. Lewis (2003). "Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices" (SEI Series in SE), AW.

[34] I. Sommerville (2000). "Software Engineering (6th Edition)". Addison Wesley.

[35] 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.

[36] Welty, C., & Ferrucci, D. A. (1999). A Formal Ontology for Reuse of Software Architecture Documents, The 1999 International Conference on Automated Software Engineering, IEEE Computer Society Press.

[37] C.Welty, "Augmenting Abstract Syntax Trees for Program Understanding", Proceedings of The 1997 International Conference on Automated Software Engineering. IEEE Computer Society Press. P. 126-133. November, 1997.

[38] W. J. Meng, J. Rilling, Y.Zhang, R. Witte and P. Charland, An Ontological Software Comprehension Process Model. In: 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). October 1st, Genoa, Italy.

[39] R.Witte, Q. Li, Y. Zhang, and J. Rilling, "Ontological Text Mining of Software Documents". 12th International Conference on Applications of Natural Language to Information Systems (NLDB 2007), June 27-29, 2007, CNAM, Paris, France.

[40] R. Witte, Y. Zhang, and J. Rilling. Empowering Software Maintainers with Semantic Web Technologies. *4th European Semantic Web Conference (ESWC 2007)*, June 3-7, 2007, Innsbruck, Austria.

[41] Witte, R., Kappler, T., Baker, C.J.O. "Ontology Design for Biomedical Text Mining". Chapter 13 in *Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences*, Springer Verlag, 2007.

[42] P. Wongthongtham, E. Chang, T.S. Dillon, I. Sommerville (2007) 'Software Engineering Ontology – Instance Knowledge Part I', International Journal of Computer Science and Network Security, USA

[43] H. J. Yang, Z. Cui, & P. O'Brien (1999). Extracting Ontologies from Legacy Systems for Understanding and Re-Engineering, 23rd International Computer Software and Applications Conference. Washington, DC, U.S.A.

[44] Y.G.Zhang, J.Rilling, V.Haarslev, "An ontology based approach to software comprehension – Reasoning about security concerns in source code". In *Proc. of 30th Int. Computer Software and Applications Conference*, 2006.

[45] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, An Ontology-based Approach for the Recovery of Traceability Links. In: *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006).* October 1st, Genoa, Italy.

[46] OWL Web Ontology Language Guide, W3C, http://www.w3.org/TR/owl-guide/ (last accessed Nov. 2007)

[47] Protégé ontology editor, http://protege.stanford.edu/ (last accessed Sep. 2007)

[48] Eclipse Java Development Tools (JDT), http://www.eclipse.org/jdt/ (last accessed Oct. 2007)

[49] Eclipse, http://www.eclipse.org (last accessed Nov. 2007)

[50] Jena, http://jena.sourceforge.net/ (last accessed Jul. 2007)

[51] Rhino, http://www.mozilla.org/rhino/ (last accessed Mar. 2007)

[52] W3C Recommendation, SPARQL Protocol And RDF Query Language (SPARQL), http://www.w3.org/TR/rdf-sparql-query/ (last accessed Nov. 2007)

[53] GATE documentation: http://gate.ac.uk/documentation/ (last accessed Jul. 2007)

[54] Infoglue, http://www.infoglue.org (last accessed Feb. 2007)

[55] Debrief, http://www.debrief.info (last accessed Mar. 2007)

[56] uDig, http://udig.refractions.net (last accessed Mar. 2007)

[57] Java Collections, http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html (last accessed Feb. 2007)

[58] uDig GIS Documentation, http://udig.refractions.net/ (last accessed Mar. 2007)

[59] uDig System, http://udig.refractions.net/confluence/display/UDIG/Home (last accessed Mar. 2007)

[60] uDig documentation, http://udig.refractions.net/docs/ (last accessed April 2007)

[61] M.Sabou, "Extracting ontologies from software documentation: a semantic method and its evaluation", *Proc. ECAI-2004 Workshop Ontology Learning and Population*, Valencia, Spain 2004

[62] Kalina Bontcheva and Marta Sabou, Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources, In *Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006).*

[63] Ian Horrocks, Oliver Kutz, Ulrike Sattler: The Even More Irresistible SROIQ. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006),* AAAI Press, 2006, pp.57-67.

An Ontological Approach for the Semantic Recovery of Traceability Links between Software Artifacts

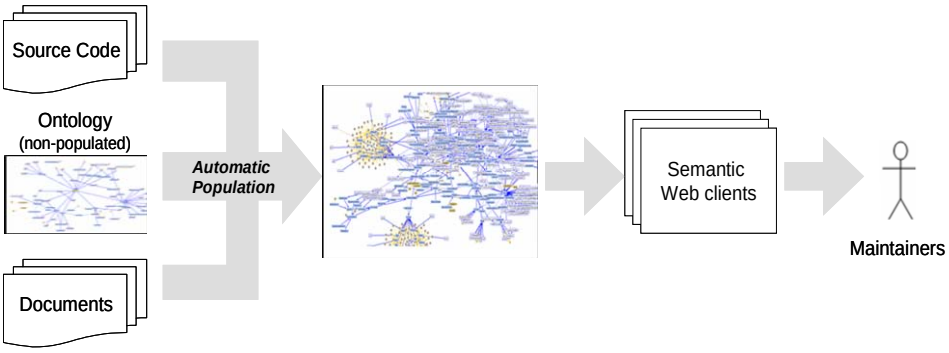

Figure 1: Ontology-Based Software Maintenance Overview

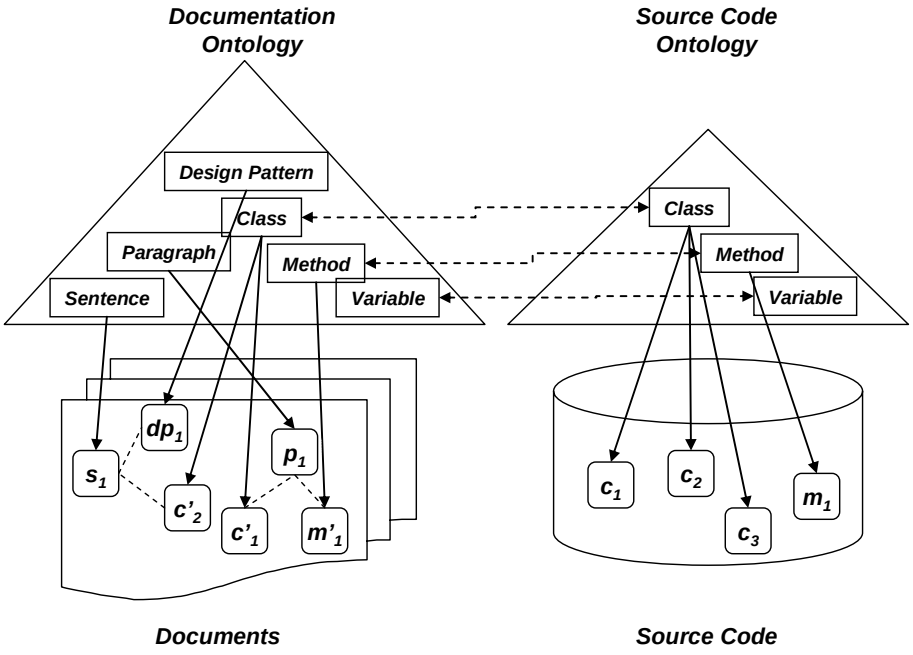

Figure 2. Linking Instances from Source Code and Documentation

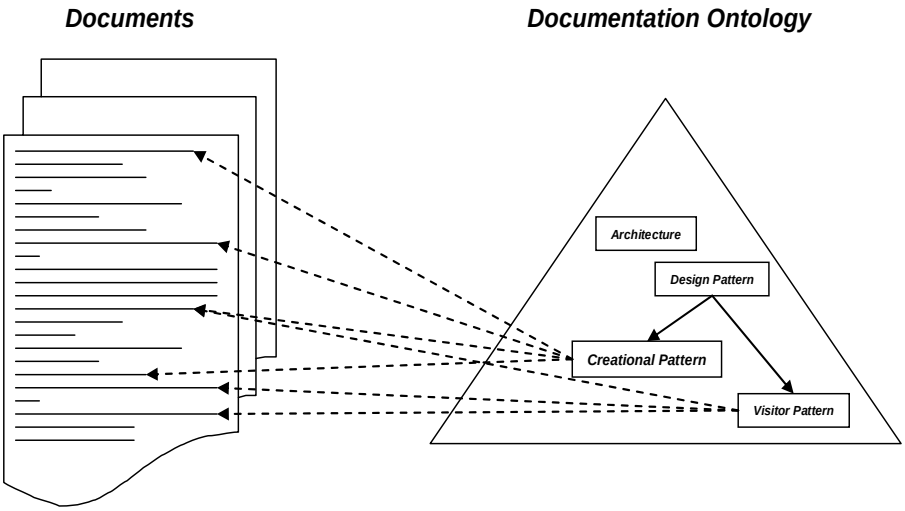

Figure 3. Classification of documentation through ontological reasoning

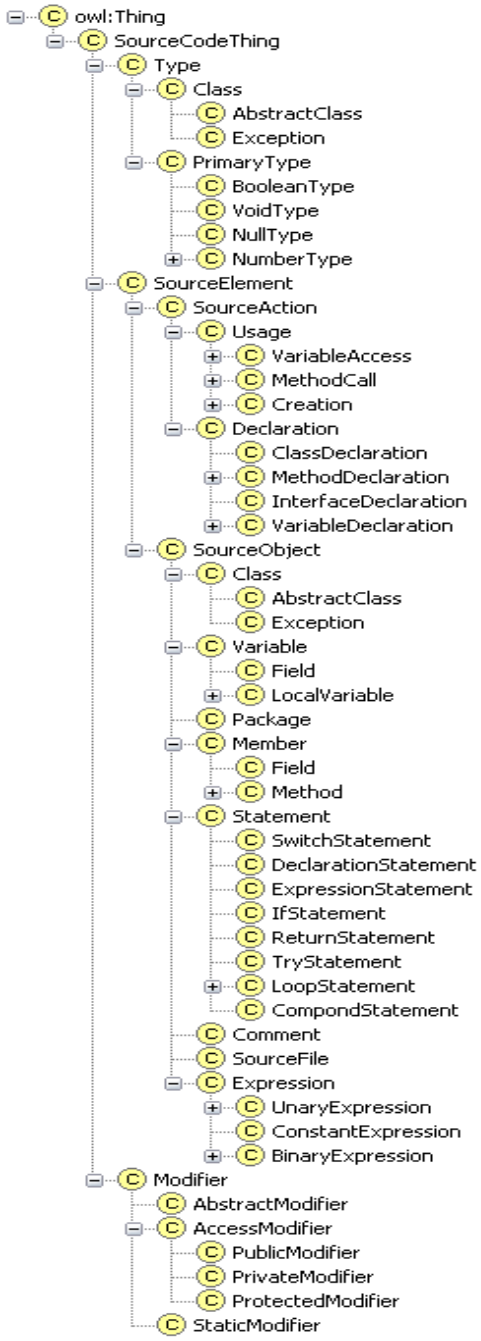

Figure 4. Concept Hierarchy in the Source Code Sub-Ontology

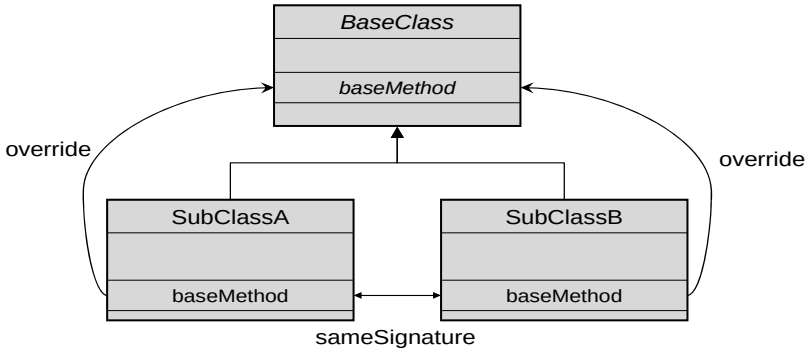

Figure 5. Static Representation of Polymorphism in OOP

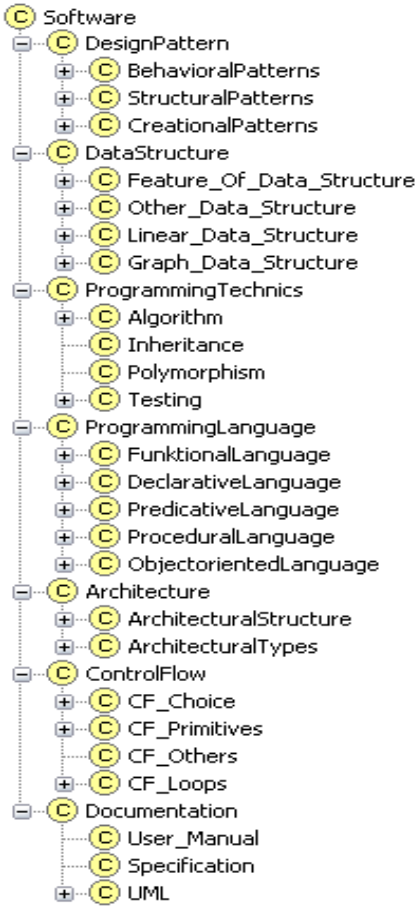

Figure 6. Top-level Concept Hierarchy in the Documentation Ontology

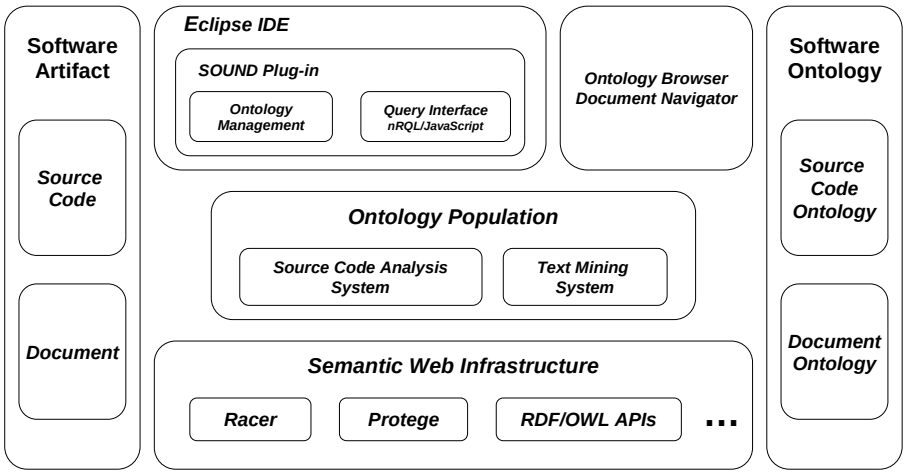

Figure 7 Semantic Web-enabled software maintenance architecture

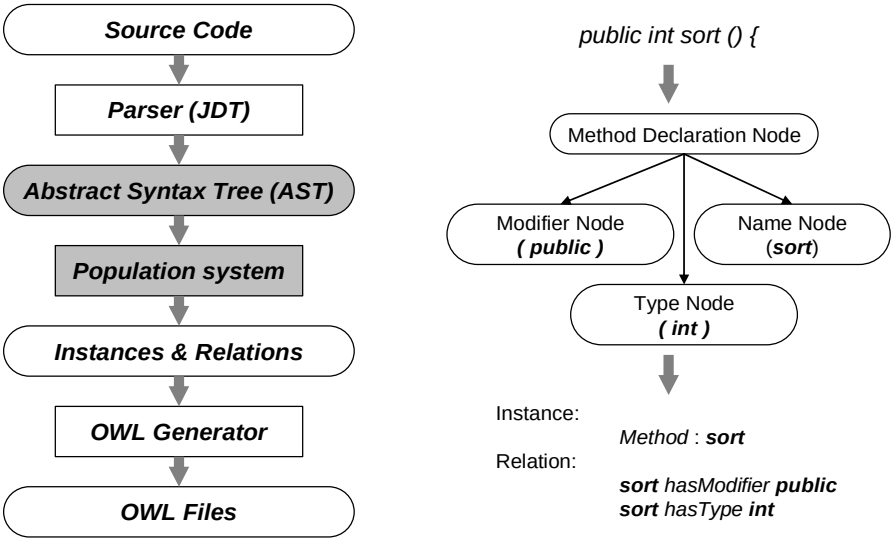

Figure 8: Automatic population of the source code ontology

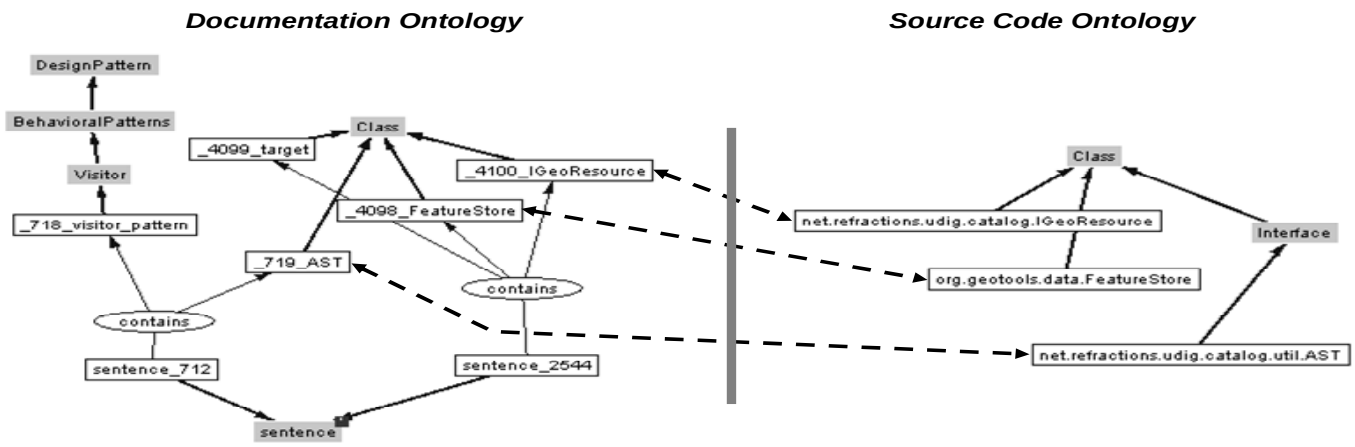

Figure 9 – Linked Source Code and Documentation Ontology