Ontology-based Program Comprehension Tool Supporting Website Architectural Evolution

Yonggang Zhang[1], René Witte[2], Juergen Rilling[1], Volker Haarslev[1]

Department of Computer Science [1] and Software Engineering
Concordia University, Montreal, Canada
{yongg_zh, rilling, haarslev}@cse.concordia.ca

Institute for Program Structures [2] and Data Organization (IPD),
Universität Karlsruhe, Germany
witte@ipd.uka.de

Abstract

A challenge of existing program comprehension approaches is to provide consistent and flexible representations for software systems. Maintainers have to match their mental models with the different representations these tools provide. In this paper, we present a novel approach that addresses this issue by providing a consistent ontological representation for both source code and documentation. The ontological representation unifies information from various sources, and therefore reduces the maintainers' comprehension efforts. In addition, representing software artifacts in a formal ontology enables maintainers to formulate hypotheses about various properties of software systems. These hypotheses can be validated through an iterative exploration of information derived by our ontology inference engine. The implementation of our approach is presented in detail, and a case study is provided to demonstrate the applicability of our approach during the architectural evolution of a website content management system.

Keywords: Program Comprehension, Software Evolution, Ontology, Automated Reasoning

1. Introduction

During the evolution of web-based applications, significant effort is spend on the comprehension of an existing system – its overall structure or the specific properties of a component. Easing the comprehension process can therefore have a major impact on reducing the cost associated with website evolution. While existing reverse engineering tools are quite successful in supporting specific comprehension tasks, they are often challenged by providing consistent and flexible software representations [1]. Software maintainers have to match their mental model of a program with the different representations provided by these tools. This overhead indicates a clear need for integrating information about different software artifacts into a uniform representation.

In this paper, we present a novel approach that addresses this issue by providing a consistent ontological representation for source code and documentation – both are primary artifacts used by maintainers during program comprehension. In addition, our approach also provides automated reasoning services supporting the comprehension process.

Our research is significant for several reasons. First, the uniform ontological representation shares common concepts between different resources, easing the integration of information, and therefore reducing the comprehension effort required for constructing mental models from software representations.

Second, representing software artifacts in a formal ontology allows maintainers to reason about various implicit properties of the software system. Taking advantage of existing ontology-based knowledge representation techniques such as Description Logics [2] and ontology reasoners [3], our approach allows users to arbitrarily define new concepts and roles (types of relations) for particular maintenance tasks and to query the ontology using either the pre- or user-defined vocabulary. Unlike previous work that utilized only informal ontologies for organizing and browsing software artifacts, a *formal ontology* (in OWL-DL) allows us to bring *automated reasoning* through DL theorem provers to the field of program comprehension, which is a significant improvement for the software engineer.

Finally, software artifacts other than source code, such as documentation, often contain rich semantic information that is rarely used by existing comprehension tools. Introducing an ontological representation allows us to utilize existing Text Mining techniques to "understand" parts of the semantics conveyed by these informal information resources.

The remainder of the paper is organized as follows: in Section 2, we discuss the role of ontology in program comprehension and presents our ontology-based program comprehension environment in detail. A case study is provided in Section 3 to demonstrate the applicability of our approach during the architectural evolution of a web site content management system. Related work is discussed in Section 4, followed by conclusions and future work in Section 5.

2. Ontology-based Program Comprehension

Program comprehension is typically referred to as the process involved in constructing an appropriate mental model of a software system to be maintained [4]. Research in cognitive science suggests that a mental model may take many forms, but its content normally constitutes an ontology [5]. Based on this observation, we present a new perspective of program comprehension, in which we specify it as an iterative process of concept recognitions and relationship discoveries in software artifacts. (Figure 1)

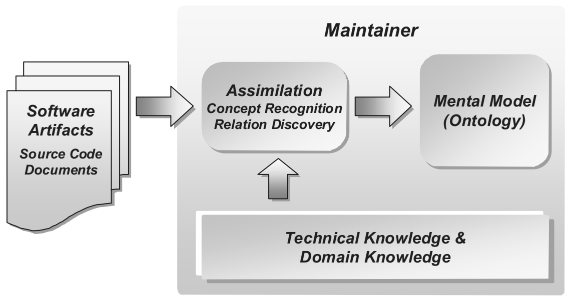

Figure 1 – Ontology-based program comprehension

Such a process typically starts with an initial mental ontology that represents a maintainer's knowledge about the programming and application domain. By reading source code and documents, instances of concepts and relationships are identified, and new concepts are discovered from the software artifacts. The result of each of these iterations therefore includes the identified instances of concepts in the program, as well as a richer ontology containing the newly discovered knowledge, i.e. a better mental model (the ontology and its instances) is constructed.

2.1 Overview

As part of this research, we have developed an ontology-based program comprehension environment – SOUND (Software Ontology for UNDerstanding), to support ontology-based program comprehension. The SOUND environment facilitates maintainers in both discovering concepts and relations within a software system, as well as automated reasoning about various properties of the software (Figure 2).

A Software Ontology has been developed to represent various software artifacts (e.g. from source code or documentation). This ontology captures major concepts and roles in the software domain. Instances of concepts and roles can be populated by our Eclipse Plug-in or Text Mining System. The discovered instances from different sources can be mapped in a semi-automated manner. Based on the software ontology, users can define new concepts/instances for particular maintenance tasks through an ontology management interface. The ontological reasoning services within the SOUND environment are provided by our ontology reasoner – Racer [3].

The following sections describe the major parts of the SOUND environment in detail – the ontology reasoner Racer, the software ontology, the text mining system, and the query interface.

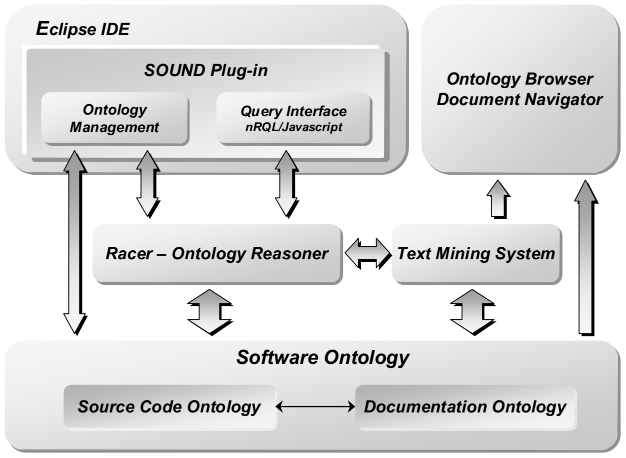

Figure 2 – Overview of SOUND environment

2.2 Description Logic and Racer

In order to formally specify our software ontology, we use Description Logic (DL) [2] as the ontology language. DL, as a knowledge representation formalism, has been long regarded as a standard ontology language. DL is also a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [6]. DL represents the knowledge of a domain by first defining the relevant concepts of the domain in a taxonomy, and then using these concepts to specify properties of individuals occurring in the domain. The use of DL allows us to formally characterize 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, having atomic concepts such as Method and Field, a new concept ClassMember can then be defined by a disjunction constructor:

$$\mathsf{ClassMember} \equiv \mathsf{Method} \sqcup \mathsf{Field}$$

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 *toString* as a method that has a *public* modifier.

$$\mathsf{toString{:}Method} \qquad \mathsf{(toString,\ public){:}hasModifier}$$

Having DL as the specification language for a formal ontology enables the use of reasoning services provided by DL-based knowledge representation systems. Our Racer system is an ontology reasoner that has been highly optimized to support very expressive DLs. Typical services provided by Racer include terminology inferences (e.g. concept consistency, subsumption, and classification, and ontology consistency) and instances reasoning (e.g. instance checking, instance retrieval, tuple retrieval, and instance realization).

For a more complete and detailed coverage of DLs and Racer, we refer the reader to [2, 3].

2.3 Text Mining and GATE

We use a *Text Mining (TM)* system to analyze documents related to a particular software system. Unlike *Information Retrieval* (IR) systems, TM 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). For example, our TM subsystem obtains information about individual software entities mentioned in the documents, like the *architecture*, its *components*, and relationships with *packages* or *classes.* These so-called *Named Entities* (NEs) are exported into the documentation ontology, which can then be loaded into a visualization tool or a reasoning system like Racer (see Figure 3).

We implemented the text mining subsystem based on the GATE *(General Architecture for Text Engineering)* framework [7], one of the most widely used NLP tools. Within the text mining process we make use of a number of standard NLP techniques. These include first dividing the textual input stream into individual tokens with a *Unicode tokeniser,* using a *Sentence Splitter* to detects 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, *Noun Phrases* (NPs) and *Verb Groups* (VGs), are created based on these tags using *chunker* modules. For more details on these steps, we refer the reader to [7] and the GATE user's manual.

2.4 Software Ontology

The software ontology in our system consists of two sub-ontologies: 1) The source code ontology represents the syntactic and semantic information of source code; 2) the documentation ontology represents semantic information extracted from software documentation.

2.4.1 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 roles needed for some specific languages (in our case, Java). Table 1 shows part of the taxonomy of the source code ontology.

Within this ontology, various roles 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.

Table 1 – Concept names in the source code ontology

Concept Name	Description and Examples
Thing	everything, top concept
JavaThing	things in Java
SourceThing	things in source code
SourceAction	actions in source, declaration, invocation, variable access, etc.
SourceObject	objects in source
Package	Java packages – *java.lang*
SourceFile	Java source files – *String.java*
Class	Java classes – *String*
Comment	inline comments – */**…*/*
Variable	variables – *System.out, temp*
Field	class variable – *System.out*
LocalVariable	local variable – *temp*
Member	class member
Field	class variable – *System.out*
Method	class method – *print(…)*
Type	types in Java – *int, float, String*
PrimaryType	primary types in Java – *int*, …
Class	abstract types – *String*

Using DL, if a role R is defined as a transitive role, and if instances of the role $(a,b)\in R$ and $(b,c)\in R$ are specified, then $(a,c)\in R$ is also implied. Transitive roles are especially useful for specifying part-of relations between source code entities (through `definedIn` role), inheritance relations between classes (through `hasSuperType` role), and indirect calling relations (through `indirectCall` role).

Concepts in the core ontology typically have a direct mapping to source code entities, and thus instances of these concepts can be automatically recognized by our SOUND plug-in, which utilizes the JDT compiler provided by Eclipse. The SOUND plug-in also identifies instances of roles (i.e. relations between source code entities) by statically analyzing the source code.

2.4.2 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.

The documentation ontology governs the identification of relevant named entities. With each concept in the ontology, a *gazetteer* list of terms is connected, which allows an *OntoGazetteer* NLP component to semantically tag individual words in the document, linking them to one (or multiple) point(s) in the ontology. Complex named entities can then be detected in another step using a cascade of finite-state transducers implementing custom grammars written in the JAPE language, which is part of GATE. For example, we can find through the OntoGazetteer that the word *layer* can be part of an architectural description. The NP analysis step will mark up the text fragment *the controller layer* as a single noun phrase, with *layer* being the head noun and *controller* a modifier, specifying exactly what layer is meant. By combining the syntactical with the semantic information, we can detect named entities, which correspond to ontology classes. Finally, additional transducers are used to discover relationships between entities, e.g., class *belongs_to* layer. These results are then used to add instances to the documentation ontology in an *ontology population* step (Figure 3).

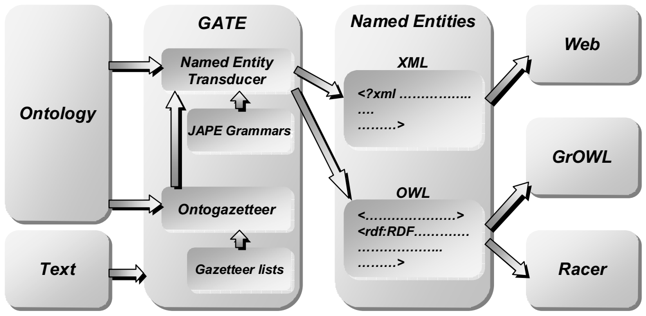

Figure 3 – Text mining in SOUND system

The documentation ontology and source code ontology share many concepts from the programming language domain, such as Package or Class. This overlap allows us to "trace" relations between source code and documentation. For example, our source code analysis tool may identify C_1 and C_2 as classes, and this information will be used by the documentation TM system to identify named entities – C_1 and C_2 and their associated information in the documents. As a result, source code entities C_1 or C_2 are linked to their occurrences in the documents, as well as other information about the two entities mentioned in the document. Users can then make ontological queries about the documents regarding properties of C_1 or C_2. For example – retrieve document passages that describe both C_1 and C_2, or retrieve documents only containing C_1 but not C_2, or is there anywhere in the documents a mention of C_1 in connection with the concept Layer.

2.5 Query Interface

Users of our SOUND environment can use the Racer query language nRQL [8] 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 even more difficult to apply. To bridge this gap between practitioners and Racer, we allow the use of a standard script language – JavaScript, as ontology query language. We provide a set of built-in functions and classes in the JavaScript interpreter – Rhino[*] to simplify user querying on the ontology. Utilizing a scriptable query language allows users to benefit from both the declarative semantics of DL as well as the fine-grained control abilities of procedural languages.

[*] available online at http://www.mozilla.org/rhino/

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 2.1 can be specified using the build-in functions as

```
ontology.define_concept("ClassMember", OR("Method", "Field"))
```

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 the query is a set of tuples that satisfy the specified restrictions. For example, the following query script retrieves all public methods:

```
var public_method_query = new Query();
public_method_query.declare("M", "P");
public_method_query.restrict("M", "Method");
public_method_query.restrict("M", "PublicModifier");
public_method_query.restrict("M", "hasModifier", "P");
public_method_query.retrieve("M");
var result = ontology.query(public_method_query);
```

This query first declares two variables M and P, and then specifies that M shall be bound to an instance of Method, and P to an instance of PublicModifier. The third restriction specifies that M and P shall have a hasModifier relation. The next statement states that this query only retrieves instances bound to M.

3 Supporting Website Architectural Evolution – A Case Study

During the evolution of web-based applications, a variety of maintenance tasks, such as architectural recovery, restructuring and component substitution, require the analysis of software systems at the architectural level. Maintainers need to comprehend the overall structure of the software system by identifying components and studying their properties. Such properties include, among others, interrelationships such as data and control communications between components, as well as internal properties, like implemented design patterns and reusability of components.

In this section, we present a case study that demonstrates how our SOUND environment can support the comprehension tasks during architectural evolution by (1) automatically identifying potential components from software documentation and (2) providing guidance for specifying them, (3) allowing maintainers to formulate hypotheses concerning various properties of the identified components, and (4) confirming or refuting these hypotheses using automated reasoning. Through each confirmation or refutation, the maintainer obtains a better understanding of the system architecture.

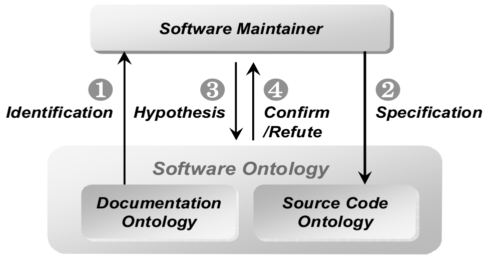

Figure 4 – Ontology-based architecture analysis process

The software under study, InfoGlue[*], is an open source Content Management System written in Java. The InfoGlue system is suitable for a wide range of applications such as public websites, portal solutions, intranets and extranets. An initial architecture document of the system is available online.

3.1 Identifying Architectural Styles and Components

The first step of an architecture analysis is typically to identify potential architectural styles [9] and candidate components in the system. Maintainers typically start the comprehension process by analyzing existing architecture documentation. Within our SOUND environment, users are referred to the populated documentation ontology to explore the architecture documents.

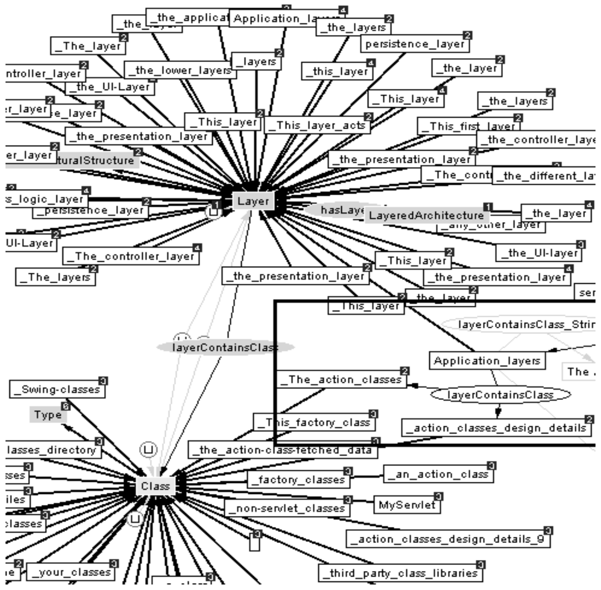

Figure 5 – Documentation ontology populated through text mining

In order to identify the architectural style of the InfoGlue system, we first use our text mining system to analyze the (architecture) documents and populate the documentation ontology. This ontology contains various concepts concerning software architecture, such as MVC, Layered Architecture, or Pipeline-and-Filters [9]. By browsing the populated ontology, we observe that a large number of instances of concept Layer (Figure 5) are discovered. This information provides us with significant clues that the InfoGlue system might be implemented using a typical Layered Architecture [9].

The discovered instances of the concept Layer are linked to their corresponding occurrences in the architecture document. Our hypothesis about the layered architecture can then be examined by further reading the architecture document with the “ontological navigation”. Within the InfoGlue architecture document, we find an architectural description that confirms our hypothesis explicitly. The informal descriptions identified in the document for these layers are summarized in Figure 6.

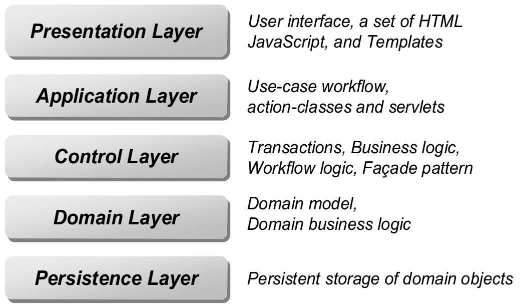
Presentation Layer
User interface, a set of HTML JavaScript, and Templates
Application Layer
Use-case workflow, action-classes and servlets
Control Layer
Transactions, Business logic, Workflow logic, Façade pattern
Domain Layer
Domain model, Domain business logic
Persistence Layer
Persistent storage of domain objects

Figure 6 – Layers in the InfoGlue system

As a result of this analysis, candidate layers as well as their informal descriptions are identified.

3.2 Specifying Components’ Properties

In order to be able to link the identified layers with their implementations, we define five instances of concept Layer within our source code ontology – *presentation_layer*, *application_layer*, *control_layer*, *domain_layer*, and *persistence_layer*. This is done by the following script:

```
ontology.add_instance(“presentation_layer”, “Layer”);
ontology.add_instance(“application_layer”, “Layer”);
ontology.add_instance(“control_layer”, “Layer”);
ontology.add_instance(“domain_layer”, “Layer”);
ontology.add_instance(“persistence_layer”, “Layer”);
```

We also use our SOUND Eclipse plug-in to analyze the InfoGlue source code to populate the source code ontology. The populated ontology contains instances of Class, Package, Method etc, as well as relations between the identified instances, such as hasSuperType, definedIn, call, and create. The populated source code ontology and the documentation ontology can then be used to identify or specify some properties of the identified layers, e.g. their containing classes or packages.

For example, the text mining system automatically discovered that the application layer contains a set of action classes, shown in Figure 7. This information provides important references for our further analysis of the documents and source code.

[*] http://www.infoglue.org

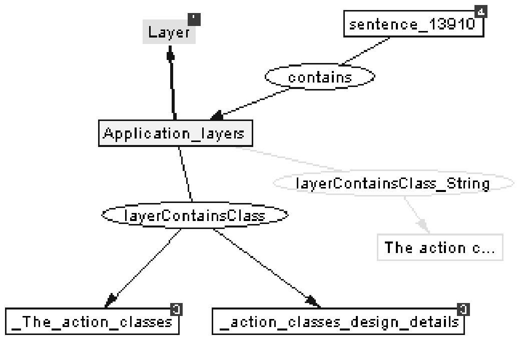

Figure 7 – Semantic information discovered by text mining

We later find that the action classes refer to classes that implement *webwork.action.Action* interface. Based on this observation, the following query can be used to retrieve all action classes and specify that all these retrieved classes are defined in the *application_layer:*

```
var query = new Query();
query.decl("C");
query.restrict("C", "Class");
query.restrict("C", "hasSuperType", "webwork.action.Action");
query.retrieve("C");
var result = ontology.query(query);

for(var i = 0; i < result.size(); i++){
    var class_name = result.get("C", i);
    ontology.add_relation(class_name, "definedIn", "application_layer");
}
```

It has to be noted that in case the newly acquired concepts are needed for other comprehension tasks, users of the SOUND environment can define these concepts in the ontology. For example, in the above analysis, the new concepts – *action class* and *super class of action class* – are acquired. The users can optionally specify them as, for example, SuperClassOfAction, and ActionClass:

```
ontology.define_concept("SuperClassOfAction");
ontology.add_instance("webwork.action.Action", "SuperClassOfAction");

ontology.define_concept("ActionClass",
                AND("Class", Exist("hasSuperType", "SuperClassOfAction")));
```

The script first creates a new concept SuperClassOfAction, and specifies that *webwork.action.Action* is an instance of that concept. Next, it creates another concept ActionClass by giving its definition, which is equivalent to the DL expression

$\text{ActionClass} \equiv \text{Class} \sqcap \exists \text{hasSuperType.SuperClassOfAction}$

Our reasoner can automatically infer instances of ActionClass by this definition. The newly created concepts then become an integrated part of the source code ontology and can be re-used for other comprehension tasks.

The other layers are linked to the source code in a similar fashion. The result of our analysis in this specification step is that each layer is semi-automatically specified by the packages and/or classes it contains.

3.3 Reasoning about Component Properties

In steps 3 and 4 of the ontology-based architecture analysis (see Figure 4), maintainers formulate hypotheses concerning various properties of the identified components, using the scriptable query language of the SOUND environment. The Racer reasoner can then be used to validate their hypotheses in a query-answer manner.

3.3.1 Control Communication

Before conducting the analysis, we hypothesized that the InfoGlue system implements a common layered architecture, in which each layer only communicates with its upper or lower layer [9]. In order to validate our hypothesis, we have performed the following queries to retrieve method calls between layers:

```
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.");
  }
}
```

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.

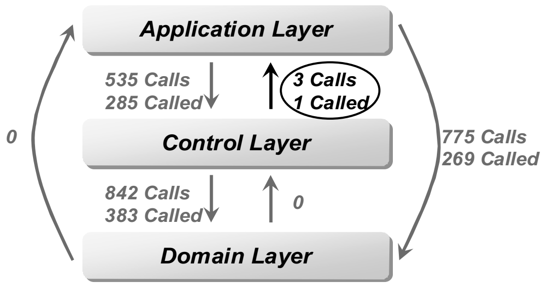

Figure 8 – Method calls between layers

Figure 8 summarizes the results of these two queries, indicating the number of method calls and the number of methods being called. The results of these queries however refute the original hypothesis about the implementation of the common layered architecture. Within the InfoGlue system, there exists significant communications from application layer to domain layer – skipping the control layer. This information is valuable for software maintainers, because it indicates that changes made in the domain layer may also directly affect the application layer.

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 be considered as either implementation defects or as the result of a special design intention. Our further inspection shows 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.

3.3.2 Data Communication

Properties of software components may also be discovered by examining the data communications between different components. In the next example, we illustrate that queries can be created similar to the one used for control communication analysis to retrieve all object creations between layers. Figure 9 summaries the results of such a query executed for the InfoGlue system. The numbers in the figure correspond to the total number of object creations (*new* expressions in Java) and the total number of classes being created.

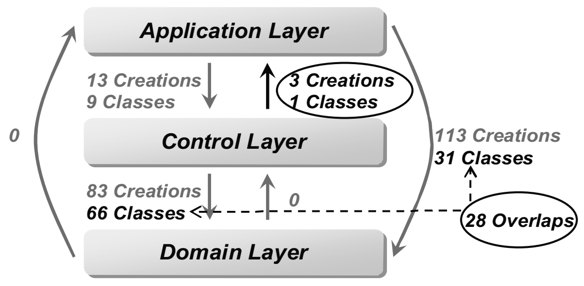

Figure 9 – Object creations between layers

Within the InfoGlue system, there are three object creations from control layer to application layer that correspond directly to the implementation defect we discovered previously.

More importantly, we also observe that the domain layer objects are created by both the control layer and the application layer. We consider that such an implementation may lead to the situation where the data integrity of the domain model can become invalidated. In order to study whether the domain object creations in the application layer can be delegated by the control layer, we execute the following query to retrieve all domain classes that have been created by *both* the application layer and the control layer:

```
var query = new Query();
query.declare("C1", "C2", "C3");
query.restrict("C1", "Class");
query.restrict("C2", "Class");
query.restrict("C3", "Class");
query.restrict("C1", "definedIn", "application_layer");
query.restrict("C2", "definedIn", "control_layer");
query.restrict("C3", "definedIn", "domain_layer");
query.restrict("C1", "create", "C3")
query.restrict("C2', "create", "C3');
query.retrieve("C3");
var result = ontology.query(query);
out.println(result.size());
```

The result of this script indicates that there are a total of 28 classes in the domain layer created by both layers. We consider that this major overlap (of the 31 classes created by the application layer) provides strong evidence that it might be possible to move the domain object creations from the application layer to the control layer. This restructuring would allow the data integrity of the domain model to be maintained by the control layer only. Such delegation also provides a cleaner implementation of the common layered architecture. We consider this information especially important for architecture re-engineering.

3.4 Case Study Summary

In this section, we presented a case study that focused on the use of our SOUND environment to support the architectural evolution of an open source web site content management system. The case study clearly demonstrates how our text mining system can automatically identify potential components and their containing source code entities, according to semantics as found in typical architecture documents. This discovered information can provide guidelines for maintenance tasks such as *architecture recovery*. Second, we detected implementation defects of architectural styles, as well as inconsistencies between documents and source code. These results can then be further used by other maintenance tasks like *architecture repairing* and *re-documenting*. In addition, we also discovered some important properties of the identified components, such as the reusability of a component. This information may contribute to the maintenance tasks such as *component substitution*. Our assumptions concerning restructuring the architecture were validated by automated reasoning. The results are also applicable for other architectural maintenance activities, like *architecture re-engineering*.

4. Related Work and Discussions

Our source code ontology and reasoning services are similar to many approaches that support queries on the source code (e.g. CIA [10], SCA [11], grok [12], RPA [13], sgrep [14], etc.). Some of these approaches that are based on the relational model to represent source code structure have some inherent problems: (1) a relational model can not capture hierarchical relations between source code elements, e.g. *field* is a special type of *variable*; (2) SQL-style query languages lack the ability to express transitivity (with exception of sgrep); and (3) different tables have to be joined many times to traverse relation paths. Recent graph-based tools (e.g. GUPRO

[15], CLG [16], etc.) overcome problems (2) and (3) of relational models by supporting transitive closures and enabling the user to formulate regular path expressions [16]. However, these approaches still can not capture type hierarchies.

Comparing with these approaches, the ontological model used in our approach supports concept hierarchies and transitive relations. In addition, the DL $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D}^-)$ supported by Racer allows the definition of role hierarchies (e.g. write is a sub-role of access) and concrete domain concepts (e.g. the line number of a source code element). The reasoning complexity of $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D}^-)$ logic is decidable.

In addition, the query language nRQL provided by Racer is more expressive than relational algebra (or non-recursive Datalog) [8]. nRQL allows for the formulation of conjunctive queries, in which query variables are used for binding to individuals that satisfy the given restrictions. Users may therefore compose complex queries to traverse relation paths by specifying relations between query variables. Moreover, in addition to the Open World Assumption within DLs, nRQL also supports localized Negation As Failure (NAF) semantics, which is especially useful for programmers to reason about the *absence* of information in a model [17].

Very little previous work exists on text mining software documents. Most of this research has focused on analysing texts at the specification level, e.g., in order to automatically convert use case descriptions into a formal representation [18, 19] or detect inconsistent requirements [20]. In contrast with these works, we aim to support the complete software documentation life-cycle, from white papers over design and implementation documents to in-line code texts (e.g., JavaDoc). To the best of our knowledge, nobody has attempted so far to automatically cross-link entities (e.g., methods, design patterns, architectures) detected by text mining software documents with corresponding entities found by source code analysis, which is an important contribution of our work.

Existing research on applying DL or formal ontology in software engineering have been addressed in early works of the LaSSIE [21] and CBMS [22] systems. 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.

In previous work, we already examined the requirements for software reverse engineering repositories [23], where we focused on dealing with incomplete and inconsistent knowledge on software artifacts obtained from different sources (e.g., conflicting information delivered by source code and document analysis), which we aim to integrate into the SOUND environment to explicitly deal with inconsistencies throughout the comprehension process

5. Conclusions and Future Work

Program comprehension is a knowledge intensive activity, requiring a large amount of effort to synthesize information obtained from different sources. In this paper, we present a novel approach that provides consistent ontological representations for various software artifacts, including source code and software documentation. Our approach aims to reduce the comprehension effort by automatically identifying concept instances and their relations in different software artifacts. The approach is based on a software ontology that also provides means to map semantic information discovered from documentation to implementation and vice versa. In addition, our approach supports the formulation of hypotheses concerning properties of a software system, and it provides ontological reasoning services to validate these hypotheses.

Formal ontologies have been long regarded as a standard technique in knowledge representation. Their underlying formalism, Description Logic, is also a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [6]. Our work promotes the use of both formal ontology and automated reasoning in program comprehension research, by providing ontological representations for various software artifacts. This representation also provides a closer mapping to a programmer's knowledge, and therefore may ease the construction of an appropriate mental model of the program under study.

Our source code and documentation ontologies are already linked via a number of shared concepts, like class or method, allowing maintainers for the first time to automatically trace entities across the code-document boundary. In order to ease understanding of large code and document bases, we aim to improve the precision of the text mining subsystem, which will allow to execute both broader and semantically richer queries, including automated reasoning spanning code, documentation, and domain knowledge. By exploring a hierarchical linking strategy, starting from code, including inline comments (like JavaDoc), over implementation, design, and specification documents to domain-specific knowledge, we will be able to offer a truly holistic process for an automated support of program comprehension.

ACKNOWLEDGEMENTS

Qiangqiang Li contributed to the text mining system.

REFERENCES

[1] D. Jin and J. R. Cordy. "A Service Sharing Approach to Integrating Program Comprehension Tools". In Proceedings of the European Software Engineering Conference, Helsinki, Finland, 2003.
[2] F. Baader et al., "The Description Logic Handbook", Cambridge University Press, 2003.

[3] V. Haarslev and R. Möller, “RACER System Description”, In Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy, Springer-Verlag, Berlin,pp. 701-705.

[4] B. Shneiderman, “Software Psychology: Human Factors in Computer and Information Systems”. Winthrop Publishers Inc. 1980.

[5] P. N. Johnson-Laird. “Mental models: towards a cognitive science of language, inference, and consciousness”. Cambridge, Mass. Harvard University Press, 1983.

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

[7] 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.

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

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

[10] Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, “The C Information Abstraction System”. IEEE Transactions on Software Engineering, 16(3):325-334, 1990

[11] S. Paul and A. Prakash, “Supporting Queries on Source Code: A Formal Framework”, Intl J. of Software Engineering and Knowledge Engineering, 1994

[12] R. C. Holt, “Binary relational algebra applied to software architecture”. Technical report, CSRI 345, University of Toronto, march 1996

[13] L. Feijs, R. Krikhaar, and R. C. Ommering, “A Relational Approach to Support Software Architecture Analysis”. Software Practice and Experience, 28(4):371-400, 1998

[14] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey, “Semantic Grep: Regular Expression + Relational Abstraction”, In Proc of the 9th Working Conference on Reverse Engineering (WCRE’02), 2002

[15] C. Lange, H. M. Sneed, and A. Winter, “Applying the graph-oriented GUPRO Approach in comparison to a Relational Database based Approach”. In Proc of the 9th International Workshop on Program Comprehension, 2001

[16] B. Kullbach, and A. Winter, “Querying as an Enabling Technology in Software Reengineering”. In Proceedings of the 3rd EuroMicro Conference on Software Maintenance and Reengineering, 1999

[17] Y. G. Zhang, J. Rilling, V. Haarslev, "An ontology based approach to software comprehension - Reasoning about security concerns in source code", in Proceedings of the 30th Annual International Computer Software and Applications Conference, 2006

[18] V. Mencl. “Deriving Behavior Specifications from Textual Use Cases”. Proc. of Workshop on Intelligent Technologies for Software Engineering (WITSE'04), Linz, Austria, Sep. 21, 2004.

[19] 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, June 15-17, 2005.

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

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

[22] 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.

[23] Ulrike Kölsch and René Witte, “Fuzzy Extensions for Reverse Engineering Repository Models”. 10th Working Conference on Reverse Engineering (WCRE), Victoria, Canada, 2003.