AN ONTOLOGY-BASED PROGRAM COMPREHENSION MODEL

Yonggang ZHANG

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

September, 2007

ABSTRACT

An Ontology-based Program Comprehension Model

Yonggang ZHANG, Ph.D.
Concordia University, 2007

Program comprehension is often regarded as a learning process where programmers actively acquire knowledge from various software artifacts to construct well formed, representative mental models of existing software systems.

In this research, we introduce a novel ontology-based program comprehension model that addresses both the challenge of knowledge acquisition and the construction of mental models. Our approach provides ontological support for program comprehension by: 1) representing various software artifacts, including source code and documents as formal ontologies; 2) utilizing ontological reasoning services to allow programmers not only to reason about properties of the software systems, but also to actively acquire and construct new concepts based on their current understanding; and 3) introducing an ontology-based comprehension model and a supporting comprehension methodology that characterize program comprehension as an iterative process of concept recognition and relationship discovery.

The research presented in this thesis is significant for several reasons. Firstly, we describe program comprehension as a process of mental ontology construction, which is directly supported by existing mental model and constructive learning theories. Secondly, we provide a unified ontological representation for various software artifacts to support the construction of mental models. Such representation allows programmers to reason about properties of the software system through concept construction and ontology exploration. Thirdly, we developed a comprehension methodology that integrates existing strategy based comprehension models into a unified knowledge acquisition framework. Finally, we present a tool implementation to support the methodology and to demonstrate the applicability of our approach.

Table of Content

List of Figures

List of Tables

List of Queries

Chapter 1. Introduction

It has been shown that 50-80% of software development resources are spent in the *software maintenance* phase (McClure, 1992), making this the most timing consuming and costly phase in the software development lifecycle. In order to perform any maintenance tasks, software systems must be understood by maintainers and therefore also making *program comprehension* an essential part of most software maintenance activities.

In order to comprehend a software system, maintainers often spend considerable time exploring and searching software artifacts, including reading source code and/or documents to locate these parts that are relevant to a specific maintenance task. Therefore, developing program comprehension tools that support the discovery and synthesis of information found in both source code and software documents are an important issue for the software maintenance research community.

Existing research in program comprehension has mainly focused so far on providing cognitive support for programmers, by representing software systems in "another form or at a higher level of abstraction" (Chikofsky & Cross, 1990). These approaches range from simple code beautifier, software visualization, to design recovery tools, etc. Common to all of these approaches is that they attempt to facilitate programmers in establishing a mapping between a user's mental model of a system and the system

model itself (Walenstein, 2002). However, two fundamental issues remain unanswered: 1) what constitutes a mental model and 2) how do programmers construct the mapping between software and the mental model?

Research in mental model theory (Johnson-Laird, 1983) suggests that the content of a mental model constitutes an *ontology*, which is regarded as a formal, explicit way of specifying the concepts and relationships in a domain of understanding (Wongthongtham, Chang, Dillon, & Sommerville, 2005). Studies in cognitive-development or constructive learning (Piaget, 1971) also regard *concepts* as building blocks of human cognition. In these theories, a learning process consists of two complementary parts: *assimilation* and *accommodation.* Assimilation incorporates new knowledge without changing prior knowledge, whereas during accommodation prior knowledge has to accommodate itself to new facts and thus these new facts will cause a reorganization of the existing knowledge. Based on these theories, program comprehension can be seen as a learning process, where programmers actively acquire knowledge from different software representations (Rajlich & Wilde, 2002). (Figure 1-1)

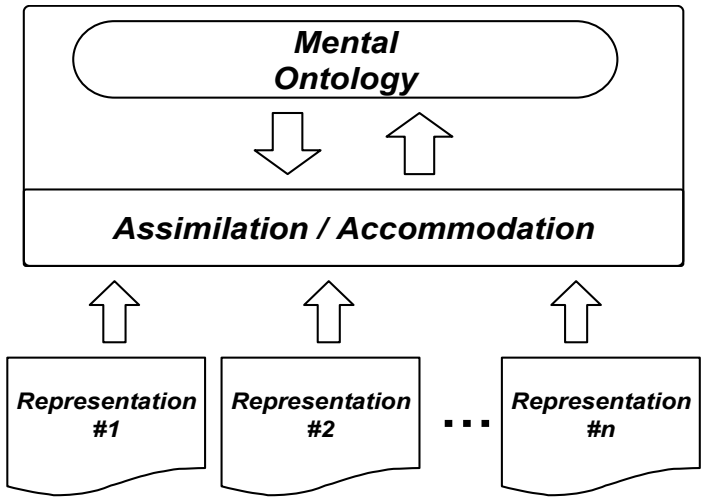

Figure 1-1 Program Comprehension as a Learning Process

An important observation made in constructive learning theories (Novak, 1998; Piaget, 1971) is that humans often actively learn new facts based on their prior knowledge and by manipulating this knowledge by using four different logical operations: conjunction, disjunction, negation, and implication. These logics operators are also commonly used in the knowledge representation community (Sowa, 1999). In particular, research in formal ontologies (Baader, 2003) provides already well defined semantics for these logic operations in the form of concept subsumption, conjunction, disjunction, and negation. State-of-the-art ontological inference engines have been developed to provide automated reasoning services to support these concept constructions.

1.1. Contribution

In this research, we introduce a novel ontology-based program comprehension approach that addresses both, the challenge of knowledge acquisition and the

construction of mental models (Figure 1-2). Our approach provides ontological support for program comprehension by:

(1) representing various software artifacts, including source code and documents as formal ontologies;

(2) utilizing ontological reasoning services to allow programmers not only to reason about properties of the software systems, but also to actively acquire and construct new concepts based on their current understanding; and

(3) introducing an ontology-based comprehension model and a supporting comprehension methodology that characterize program comprehension as an iterative process of concept recognition and relationship discovery.

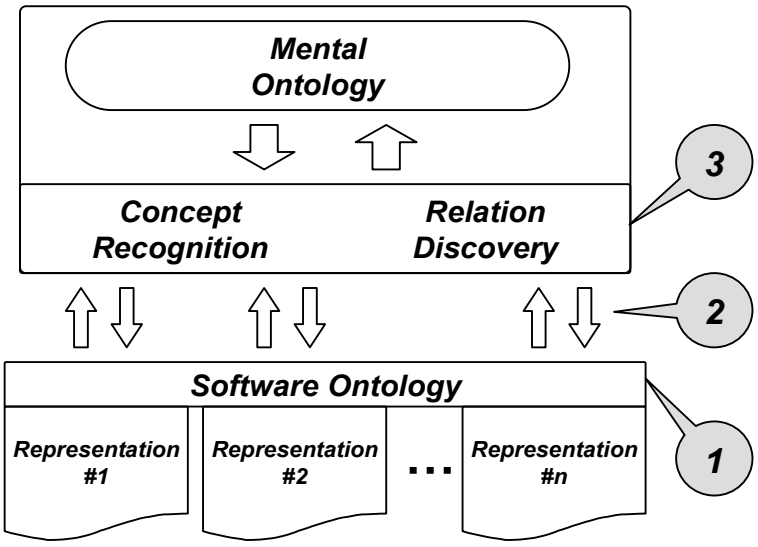

Figure 1-2 Ontology-based Program Comprehension

The research presented in this thesis is significant for several reasons. Firstly, we describe program comprehension as a process of mental ontology construction, which is directly supported by existing mental model and constructive learning theories. Secondly, we provide a unified ontological representation for various software artifacts, to support the construction of mental model. Such representation allows programmers to reason about properties of the software system through concept construction and ontology exploration. Thirdly, we developed a comprehension methodology that integrates existing strategy based comprehension models into a unified knowledge acquisition framework. Finally, we present a tool implementation to support the methodology and to demonstrate the applicability of our approach.

1.2. Publications

Research results presented in this thesis have been published in proceedings of several recognized international journals, conferences and workshops, including

- Proceedings of the 30th IEEE International Computer Software and Applications Conference (COMPSAC'06), where we proposed our ontology-based program comprehension framework, and demonstrated its applicability in security analysis (Zhang, Rilling, & Haarslev, 2006).

- Proceedings of the 8th IEEE International Symposium on Web Site Evolution (WSE'06), where we presented the initial implementation of our approach and

provided a comprehensive case study for supporting web site evolution (Zhang, Witte, Rilling, & Haarslev, 2006).

- Proceedings of the 2nd IEEE International Workshop on Software Evolvability (Evol'06), where we discussed the application of formal ontology and reasoning in software development process management (Meng, Rilling, Zhang, Witte, Mudur, & Charland, 2006).

- Proceedings of the 3rd International Workshop on Meta-models, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM'06), where we focus on a specific software engineering task - Traceability Links, and discussed how our approach in combination with source code analysis and text mining techniques may facilitate maintainers to recover links between source code documents (Zhang, Witte, Rilling, & Haarslev, 2006). This paper received the best paper award.

- Proceedings of the 3rd International Workshop on Meta-models, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM'06), where we presented another paper that proposed an ontology-based framework for software development process management (Meng, Rilling, Zhang, Witte, & Charland, 2006). This paper received also the best paper award.

- Proceedings of the 9th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, where we extended and combined the two

papers published in ATEM'06 (Rilling, Zhang, Meng, Witte, Haarslev, & Charland, 2006).

- Proceedings of the International Symposium on Grand Challenges in Traceability (GCT'07), where we addressed several challenges in traceability research, and illustrated how our ontology-based approach can automatically recover traceability links and classify documents (Rilling, Witte, & Zhang, 2007).

- Proceedings of the 4th European Conference on Semantic Web (ECSW'07), where we presented the detailed implementation of this research in the background of Semantic Web, with focus on ontology population (Witte, Zhang, & Rilling, 2007). This paper received the best paper award.

- Proceedings of the 12th International Conference on Applications of Natural Language to Information Systems (NLDB'07), where we focused on text mining aspect of this research, and presented a complete evaluation in terms of precision and recall (Witte, Li, Zhang, & Rilling, 2007).

- IET Software Journal, Special Issue on Natural Language in Software Development (to appear), where we presented the ontology-based program comprehension model with focus on traceability link recovery (Witte, Li, Zhang, & Rilling).

Parts of this research are a joint effort with Dr. Rene Witte from the Institute for Program Structures and Data Organisation (IPD), Faculty of Informatics, University of Karlsruhe, Germany. His contribution mainly focused on utilizing information obtained

from the source code ontology to perform text mining on software documents, whereas on the source code analysis, ontology design, automated reasoning, and integrating information from both source code and software documents were performed as part of the research presented in this thesis.

The remainder of this thesis is organized as follows: in Chapter 2, we introduce background relevant to our research, namely program comprehension and Description Logics. In Chapter 3, our ontology-based comprehension model, the methodology, a knowledge base, and several applications are presented. Chapter 4 discusses the implementation of our approach, followed by its evaluation in Chapter 5. Conclusions and future work are discussed in Chapter 6.

Chapter 2. Research Background and Related Work

This research addresses an important issue in software engineering – program comprehension, which involves a large amount of fine-grain activities to recognize relevant concepts and to identify their relationships. In this chapter, relevant research background is introduced, including various existing approaches to facilitate source code comprehension, and ontologies - a knowledge representation approach to organize concepts and their relationships.

2.1. Program Comprehension

In what follows, we review related research in program comprehension, including earlier efforts in modeling cognitive aspects of program comprehension, classification of program comprehension tools that can be used to assist software maintainers, as well as a discussion on open research challenges in this area.

2.1.1. Program Comprehension Model

There exists quite a significant body of research on modeling program comprehension in terms of mental representations and processes for creating these representations. These models can be categorized in three major theories – bottom-up models, top-down models, and integrated models.

Bottom-up Model

Bottom-up theories in program comprehension consider that understanding of a program is built from the bottom-up, by reading source code and then mentally *chunking* or *grouping* these statements into higher level abstractions. These abstractions are further aggregated until high-level understanding of the program is obtained (Shneiderman, 1980).

Shneiderman and Mayer's bottom-up model (Shneiderman & Mayer, 1979) differentiates between syntactic and semantic knowledge of programs. Syntactic knowledge is language dependent and focuses on statements and basic units in a program. Semantic knowledge on the other hand is language independent and is built progressively until a mental model can be formulated that describes the application domain. The final mental model is thus obtained through the chunking and aggregation of semantic components and syntactic fragments of source code.

Pennington's model (Pennington, 1987) is also based on a bottom-up theory. She investigated in her work, the role of programming knowledge and the nature of mental representations in program comprehension. She observed that programmers first develop a control flow abstraction of the program that captures the sequence of operations in the program. This abstraction is referred to as the program model and is developed through the chunking of microstructures in the text (statements, control constructs and relationships) into macrostructures (text structure abstractions or chunks) and through cross-referencing these structures. Once the program model has been fully

assimilated, the situation model is developed. The situation model consists of knowledge about data-flow abstractions (changes in the meaning or values of program objects) and functional abstractions (the functionalities in the application domain). The resulting situation model requires knowledge of the application domain and it is also built from the bottom-up.

Top-down Model

Top-down theories in program comprehension assume that the comprehension process is based on first reconstructing knowledge about the problem domain and then mapping this domain knowledge to the actual source code (R. Brooks, 1983). The top-down comprehension process starts with a hypothesis concerning the global view of the program. The initial hypothesis is refined in a hierarchy by formulating subsidiary hypotheses. The confirmation/refutation of hypotheses depends on the presence/absence of *beacons*. A beacon is a set of features that indicates the existence of hypothesized structures or operations, for example a function called *swap* in a sorting program. The discovery of a beacon permits code features to be *bound* to hypotheses (R. Brooks, 1983).

Soloway and Ehrlich (Soloway & Ehrlich, 1984) also observed that top-down understanding is used when the code or type of code is familiar to programmers. Expert programmers use two types of programming knowledge during program comprehension (Soloway & Ehrlich, 1984). *Programming plans* are generic fragments of code that represent typical scenarios in programming. For example, a sorting program

will contain a loop which compares two numbers during each iteration. *Rules of programming discourse* capture the conventions of programming, such as coding standards and algorithm implementations.

The understanding of a program is built top down by forming a hierarchy of goals and programming plans. Rules of programming discourse and beacons help decompose goals and plans into lower level goals and plans (Soloway & Ehrlich, 1984).

Integrated Model

Mayrhauser and Vans's work (Von Mayrhauser & Vans, 1995) integrates Soloway's top-down model with Pennington's bottom-up model. During their experiments they observed that some programmers frequently switching between these comprehension models. The *Integrated Metamodel* (Von Mayrhauser & Vans, 1995) is based on this observation and consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform the comprehension process:

- The *top-down (domain) model* is usually invoked when the programming language or code is familiar. It incorporates domain knowledge that describes program functionality as starting point for formulating hypotheses. The top down model is usually developed using an opportunistic or as-needed strategy.

- The *program model* may be invoked when the code and application is completely unfamiliar. The program model is a control flow abstraction, and may be developed as an initial mental representation.

- The *situation model* describes data-flow and functional abstractions in the program. A situation model may be developed after a partial program model has been formed using systematic or opportunistic understanding strategies.

- The *knowledge base* consists of the information needed to build these three cognitive models. It refers to initial knowledge that the programmer has before the maintenance task and is used to store new and inferred knowledge.

In the Integrated Metamodel, understanding is built at several levels of abstraction simultaneously by switching between top-down or bottom-up processes, and any of the processes may be activated at any time (Von Mayrhauser & Vans, 1994).

Letovsky (Letovsky, 1986) also considers that comprehension occurs either top-down or bottom-up depending on the cues available. He considers programmers as opportunistic processors capable of exploiting either bottom-up or top-down cues. There are three components in his model:

- The *knowledge base* encodes a programmer's expertise and background knowledge. The programmer's knowledge may consist of application and programming domain knowledge, program goals, a library of programming plans and rules of discourse.

- The *mental model* encodes a programmer's current understanding of a program. Initially the mental model consists of a specification of the program goals. It later evolves into a mental model which describes the implementation in terms of data structures and algorithms used. The final mental model includes a mapping from the specified program goals to the relevant parts of the implementation.

- The *assimilation process* describes how the mental model evolves using a programmer's knowledge base, source code, and documentations. The assimilation process may be a bottom-up or top-down process depending on the programmer's initial knowledge base. Inquiry episodes are the central activity during the assimilation process with inquiry episodes corresponding to a programmer asking a question (e.g. what is the purpose of variable x), guessing an answer (x stores the maximum of a set of numbers), and then searching through code and documentation to verify the answer.

2.1.2. Program Comprehension Tools

A variety of tools for assisting program comprehension have been proposed. The general goal of these tools is to provide means for maintainers to understand a software system from different functional and behavioral perspectives (Tilley, Paul, & Smith, 1996). Functional perspective provides insights into the "what" functionalities a system provides. Behavioral perspective focuses on the other hand on the "how" a system accomplishes its functionalities.

Program comprehension tools support various activities that are characteristics of the program comprehension process, including among others (Jin & Cordy, 2003):

- Decomposition: Breaking larger systems into subsystem components.
- Pattern Matching: Identification of instances in the source code where a certain pattern occurs
- Dependency Analysis: Evaluation of the reliance of different components in terms of control, data, etc.
- Metrics: Measurement of the program according to accepted standards for various characteristics such as size, complexity, quality, etc.
- Exploration: Support for navigation and searching throughout the program artifacts.
- Simplification: Representing software system in high-level abstractions or isolating source code by certain criteria.
- Visualization: Providing visualized representation of software system with metaphors that associate visual objects with source code.

Several conceptual frameworks (Nelson, 1996) (Rugaber, 1995) (Storey, 2005) for the classification of software engineering tools that support program comprehension have

been proposed. Some of the more prominent approaches are summarized in what follows.

- *Textual, lexical and syntactic analysis* – these approaches focus on the source code and its representations. Most of these tools are based on static analysis. In these approaches, source code is parsed and stored into an intermediate format (e.g. Abstract Syntax Tree). This intermediate representation is then used to create various levels of abstraction or to support different types of analysis, such as Program Slicing (Weiser, 1984), Clichés (Wills, 1994), Design Patterns (Antoniol, Fiutem, & Cristoforetti, 1998; Keller, Schauer, Robitaille, & Lague, 2001) Concept Locations (Marcus, Rajlich, Buchta, Petrenko, & Sergeyev, 2005), Concerns (Revelle, Broadbent, & Coppit, 2005) etc.

- *Execution and testing* – There exists a variety of methods for profiling, testing and observing program behavior, including actual execution (Weiser, 1981) and inspection walkthroughs (Pagan, 1991). Recent approaches mainly focus on component communications (Hendrickson, Dashofy, & Taylor, 2005), thread/process communications (Reiss, 2005), and execution trace analysis (Fischer, Oberleitner, Gall, & Gschwind, 2005) etc.

- *Graphing method* – including earlier approaches such as graphing the control flow of the program (Hecht, 1977), the data flow of the program (Hecht, 1977), program dependence graphs (Ferrante, Ottenstein, & Warren 1987). Recent approaches also try to visualize software systems in higher-level of abstractions

(e.g. UML (Sun & Wong, 2005), architectural communication (Murphy, Notkin, & Sullivan, 1995; Synytskyy, Holt, & Davis, 2005)) or complex textual views of source code (Cox & Collard, 2005).

- *Domain knowledge based analysis* – these approaches focus on recovering the domain *semantics* of programs by combine domain knowledge representation and source code analysis. DESIRE (Biggerstaff, Mitbander, & Webster, 1993) and LaSSIE (Devanbu, Brachman, Selfridge, & Ballard, 1990) systems are typical examples in this category. Recently, there is a trend to use techniques from other domain (e.g. Data Mining (Tjortjis, Sinos, & Layzell, 2003) Concept Analysis (Antoniol, Casazza, di Penta, & Merlo, 2001), and Information Retrieval (Antoniol, Canfora, Casazza, De Lucia, & Merlo, 2000) etc.) to discover domain specific concepts in software systems.

2.1.3. Program Comprehension Research Challenges

Although research in program comprehension has made some significant progress in the last two decades, there still remain several major open challenges. In this section, we briefly discuss some of these challenges that are closely related to our research.

- ***Cognitive Theories***

Program comprehension is a cross-disciplinary research domain that not only involves software engineering, but also includes psychological, cognitive and learning aspects of programmers' behaviors. Often, research methods and theories applied in the program

comprehension domain were adopted from these cross disciplinary areas to develop comprehension models and tools. Perceived benefits of introducing these theories are that they provide rich explanations of how to analyze and model a programmer's performance, and may lead to more efficient process and methods in the software engineering domain (Hohmann, 1996).

However, as the programming paradigm and software development processes changes, richer cognitive theories need to be further developed. For example, learning theories (Exton, 2002) will become more relevant to this area (Rajlich & Wilde, 2002; Storey, 2005) ; the social and organization aspects of program comprehension are currently being investigated (Gutwin, Penner, & K. Schneider, 2004); research is conducted to study the understanding of object-oriented programming from behavioral perspective (Walkinshaw, Roper, & Wood, 2005) etc.

- ***Tool Integration***

Although there has been much progress made towards improving the performance and usefulness of program comprehension tools, most of these tools continue to exist in isolation, lacking any effective means for sharing information among each other. Software maintainers typically work independently with each tool starting from scratch, and manually integrating results from different tools. (Ebert, Kullbach, & Winter, 1999)

An additional requirement for program comprehension tool integration is that most of the existing tools have their own specific strength or specialized application areas but

are weak in other areas (Jin & Cordy, 2003), leading to a situation there no single tool exists to provide all the required functionality and flexibility to support a maintainer's software comprehension needs. Therefore, research attention has been focused on enabling reverse engineering tools to interoperate with each other.

However, such integration is inherently difficult. Jin et al (Jin & Cordy, 2003) consider the primary barrier for tool interoperability is caused by the differences in the representation of software knowledge that each tool maintains. These differences are both at the syntactical and semantic level. The syntactical variations are caused by structure difference and the way information is manipulated and stored by each tool. These structural differences can be reconciled through representational mapping and data translation. Semantic differences are more difficult to resolve due to a myriad of semantic variations among models for the various programming paradigms, as well as differences caused through application domain knowledge that has to be modeled.

- ***Use of Domain Knowledge***

A common approach for understanding software is to establish a global picture of a system/sub-system by using, e.g. UML class diagram, ER diagram. The goal is to reflect the physical and logical structure of the components and their inter-communications. However, this approach is often insufficient to fully comprehend the *purpose* (Biggerstaff et al., 1993) of a given piece of code. One of the possible reasons, as Beck and Johnson discussed, is that "existing design notations focus on communication of the

what of the designs, but almost completely ignore the *why*" (Beck & Johnson, 1994; Keller, Schauer, Robitaille, & Page, 1999).

In order to fully understand programs, the application domain must be well understood and, preferably, modeled. Once recognized, a domain can be characterized by its terminology, common relationships, architectural approach and literature. Domain models provide reverse engineers with a set of expected constructs to look for in the code. These might be computer representations of real world objects such as company or employer or they may correspond to algorithms, such as the LIFO method of appraising inventories. Domain models might also represent overall architectural schemes, such as n-tiers architecture for implementing a web based application. Objects in a domain model are related to each other and organized in prototypical ways that are likely to be recognized in the later program implementation. Hence, a domain representation acts often as a schema for controlling the reverse engineering process and as a template for organizing its result. Incorporating domain information into a program comprehension process is essential to answering the why questions. (Rugaber, 1995)

The challenge of helping maintainers to construct associations between source code and its functionality, with regard to large software system, is caused mainly by the difficulty in modelling the domain itself and the inadequacy of theories/techniques to establish the relationship between software implementation and domain model.

2.2. Ontology and Description Logics

One of the motivations of this research is to utilize and apply ontologies and Description Logics as the underlying representation formalism to discover implicit facts in source code and documentation. In this section, we introduce related background on Description Logics and its notations.

2.2.1. Description Logics Overview

Existing approaches in knowledge representation can be roughly divided into logic and non-logic based formalisms. The logic-based approaches use predicate calculus to capture facts. In contrast, the non-logic-based representations model knowledge by means of some ad-hoc data structures and perform reasoning to manipulate these structures. Among these non-logical approaches, *semantic networks* (Quillian, 1967) and *frames* (Minsky, 1974) are considered as *network structures*, where the content of the network represents sets of individuals and their relationships. Unfortunately, both approaches lack a precise semantic characterization, resulting in a situation where every system based on semantic network or frame may behave differently from the others. (Baader, 2003)

One important observation towards providing a formal semantics for these *semantic networks* and *frame* based representations is that frames can be characterized by first-order logic (P. J. Hayes, 1979). Using first-order logics, sets of individuals can be denoted as unary predicates, and relationships between individuals are denoted as binary predicates. It has been shown that frames and semantic networks do not require all the

power of first-order logic, rather only some fragments of it (Brachman & Levesque, 1985). Furthermore, the forms of reasoning used typically in network-based representations can also be accomplished by specialized reasoning techniques without necessarily requiring first-order logic theorem provers. As a result of these observations, Description Logics was established as a knowledge representation formalism.

Description Logic (DL) is the most recent category for a family of knowledge representation formalisms that represent knowledge of an application domain by first defining the relevant concepts of the domain and then using these concepts to specify properties of objects and individuals occurring in the domain (Baader, 2003). Earlier work in DLs focuses on the representation languages that are used to establish the basic terminology adopted in the modeled domain, and then the set of concept-forming constructs admitted in the language. Recently, as the studies of the underlying logical formalisms become more and more mature, researches in DLs mostly focus on developing efficient and expressive DLs reasoners, e.g. Racer (Haarslev & Möller, 2003), FaCT (Tsarkov & Horrocks, 2006), Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2006). Other promising research is the application of DLs in various domains, such as conceptual modeling, database, natural language processing, medical information, etc.

Unlike their predecessors – *semantic networks* and *frames*, DLs have a formal, logic-based semantics. Another DL specific feature is the emphasis on reasoning as a central service – classification of concepts and individuals. Classification of concepts determines subsumption relationships between the concepts of a given terminology, and thus

allows one to structure the terminology in the form of a subsumption hierarchy. Classification of individuals determines whether a given individual is an instance of a certain concept, and thus provides useful information on the properties of an individual.

Since DL emphasizes inference services, investigating the computational complexity of a given DL inference problem becomes an important issue. Decidability and complexity of inference problems depend on the expressive power of DL – very expressive DLs are likely to have inference problems of high complexity, sometimes even undecidable; while very weak DLs may not be sufficiently expressive to represent the important concepts of a given domain (Baader, 2003). As a result, studying the trade-offs between expressivity and complexity has been a major issue of ongoing DL research.

2.2.2. Syntax and Semantics

Different DL variations can be distinguished from each other through concept and role constructors supported. Among them, $\mathcal{AL}$ (Attributive Language) (Schmidt-Schauß & Smolka, 1991) has been considered as the minimal language that is of practical interest. Most of the other DL languages are extensions of $\mathcal{AL}$.

Concept constructors in $\mathcal{AL}$ are formed according to the following syntax rules:

C, D →	A	|	(atomic concept)
	$\top$	|	(universal concept)
	$\bot$	|	(bottom concept)

$\neg A$	|	(atomic negation)
$C \sqcap D$	|	(intersection)
$\forall R.C$	|	(value restriction)
$\exists R.\top$		(limited existential quantification)

where A and B are atomic concepts, C and D are concept descriptions, and R is atomic role.

While the syntax may have different flavors in different settings, the semantics of DLs is typically given by Tarski-style semantics. We consider an *interpretation* $\mathcal{I}$ that consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to concept descriptions by the following inductive definitions:

$\top^{\mathcal{I}}$	$=$	$\Delta^{\mathcal{I}}$
$\bot^{\mathcal{I}}$	$=$	$\varnothing$
$(\neg A)^{\mathcal{I}}$	$=$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
$(C \sqcap D)^{\mathcal{I}}$	$=$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$(\forall R.C)^{\mathcal{I}}$	$=$	$\{ a \in \Delta^{\mathcal{I}} \mid \forall b.(a, b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}} \}$
$(\exists R.\top)^{\mathcal{I}}$	$=$	$\{ a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in R^{\mathcal{I}} \}$

More expressive languages are obtained by adding further constructors to $\mathcal{AL}$. For example, following constructors:

$C \sqcup D$	(union of concepts)
$\exists R.C$	(full existential quantification)
$\geq n\ R$	(at-least number restriction)
$\leq n\ R$	(at-most number restriction)
$\neg C$	(full negation)

extend $\mathcal{AL}$ to $\mathcal{ALCN}$. The semantics of these constructors are given as:

$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

$$(\exists R.C)^{\mathcal{I}} = \{ a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \}$$

$$(\geq n\ R)^{\mathcal{I}} = \{ a \in \Delta^{\mathcal{I}} \mid |\{ b \mid (a, b) \in R^{\mathcal{I}} \}| \geq n \}$$

$$(\leq n\ R)^{\mathcal{I}} = \{ a \in \Delta^{\mathcal{I}} \mid |\{ b \mid (a, b) \in R^{\mathcal{I}} \}| \leq n \}$$

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

In addition, since roles are interpreted as binary relations in DLs, it is also natural to extend above languages by employing common operations on binary relation as role constructors. In this thesis, the following role constructors are adopted.

$$(R^{+})^{\mathcal{I}} = (R^{\mathcal{I}})^{+} \quad \text{transitive role}$$

$$(R^{-})^{\mathcal{I}} = \{(b, a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (a, b) \in R^{\mathcal{I}}\} \quad \text{inverse role}$$

2.2.3. Description Logic System

A DL based knowledge representation system provides typical facilities to set up knowledge bases and to reason about their content (Baader, 2003). In what follows (Figure 2-1), we illustrate a typical Description Logic System.

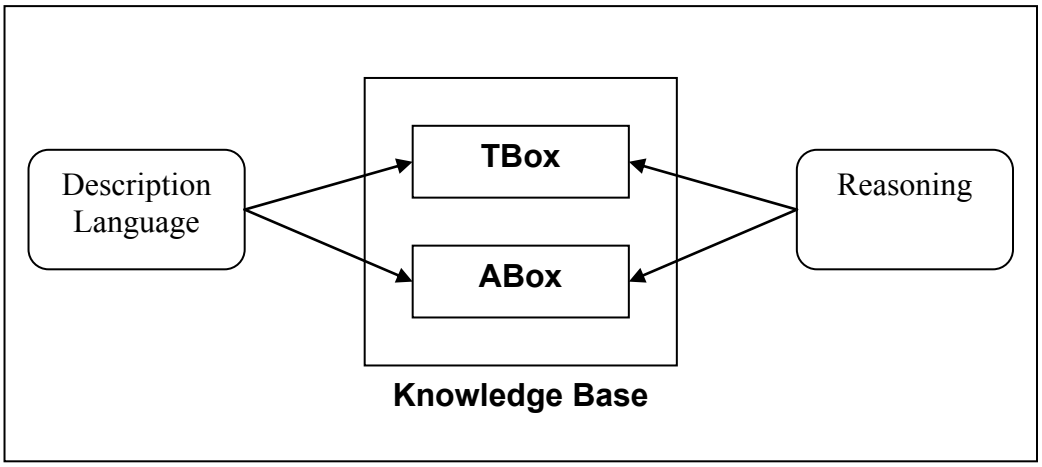

Figure 2-1 Description Logic System

Such a knowledge base (KB) consists of two components – the TBox contains the *terminology*, i.e. the vocabulary of an application domain, and the ABox contains *assertions* about named individuals in terms of this vocabulary.

The terminology is specified using description languages introduced previously in this section, as well as *terminological axioms*, which make statements about how concepts or roles are related with each other. In the most general case, terminological axioms have the form

$$C \sqsubseteq D \ (R \sqsubseteq S) \text{ or } C \equiv D \ (R \equiv S)$$

where C and D are concepts (R and S are roles). The semantics of axioms is defined as: an interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ ($R \sqsubseteq S$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ ($R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$). A Tbox, denoted as $\mathcal{T}$, is a finite set of such axioms.

The assertions in an ABox are specified using concept assertions and role assertions, which have the form

$$C(a),\ R(a,\ b)$$

where C is a concept, R is a role, and a, b are names of individuals. The semantics of assertions can be given as: the interpretation $\mathcal{I}$ satisfies the concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and it satisfies the role assertion $R(a, b)$, if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. An Abox, denoted as $\mathcal{A}$, is a finite set of such assertions.

A Description Logic system not only stores terminologies and assertions, but also offers services that allow to *reason* about them. Typical reasoning services for a TBox are to determine whether a concept is *satisfiable* (i.e. non-contradictory), or whether one concept is more general than another one (i.e. *subsumption*). Important reasoning services for an ABox are to find out whether its set of assertions is *consistent*, and whether the assertions in an ABox entail that a particular individual is an *instance* of a given concept description.

A DL knowledge base might be embedded into an application, in which some components interact with the KB by querying the represented knowledge and by modifying them, i.e. by adding and retracting concepts, roles, and assertions. However, many DL systems, in addition to providing an application programming interface that

consists of functions with a well-defined logical semantics, provide an escape hatch by which application programs can operate on the KB in arbitrary ways. (Baader, 2003)

2.2.4. Racer and nRQL

Racer (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner) (Haarslev & Möller, 2003) is a knowledge representation system that is highly optimized for a very expressive DL – $\mathcal{ALCQHI}_{R+}$, also known as $\mathcal{SHIQ}$ (Ian Horrocks, Sattler, & Tobies, 2000). The $\mathcal{SHIQ}$ logic is basic logic $\mathcal{ALCN}$ augmented with role hierarchies, qualified cardinality restrictions, inverse roles, and transitive roles.

In addition to these basic features, Racer also facilitates algebraic reasoning including concrete domains for dealing with min/max restrictions over the integers, linear polynomial (in-)equations over the reals or cardinals with ordered relations, and equalities and inequalities of strings. Racer also supports the specification of general terminological axioms. A TBox may contain general concept inclusions, which state the subsumption relationship between two concept terms.

Given a TBox, various kinds of conceptual queries can be answered, including –

- Concept consistency w.r.t. a TBox: is the set of individuals described by a concept forced to be empty?
- Concept subsumption w.r.t. a TBox: is there a subset relationship between the set of individuals described by two concepts?

- Find all inconsistent concept names in a TBox: Inconsistent concept names result from TBox axioms, and it is very likely that they are the result of modeling errors.

- Determine the parents and children of a concept w.r.t. a TBox.

If also an ABox is given with regard to a TBox, among other, the following types of queries are supported by Racer –

- Check the consistency of an ABox w.r.t. a TBox: Are the restrictions given in an ABox w.r.t. a TBox too strong, i.e. do they contradict each other? Other queries are only possible w.r.t. consistent ABoxes.

- Instance testing w.r.t. an ABox and a TBox: Is the object for which an individual stands a member of the set of objects described by a specified concept? The individual is then called an instance of the concept.

- Instance retrieval w.r.t. an ABox and a TBox: Find all individuals from an ABox such that the objects they stand for can be proven to be a member of a set of objects described by a certain query concept.

- Retrieval of tuples of individuals (instances) that satisfy certain conditions w.r.t. an ABox and a TBox.

- Computation of the direct types of an individual w.r.t. an ABox and a TBox: Find the most specific concept names from a T-box of which a given individual is an instance.

- Computation of the fillers of a role with references to an individual w.r.t. an ABox and a TBox.

- Check if certain concrete domain constraints are entailed by an ABox and a TBox.

In addition, Racer also provides a semantically well-defined query language – nRQL (new Racer Query Language) (Haarslev, Moller, & Wessel, 2004), which also supports negation as failure (NAF), numeric constraints w.r.t. attribute values of different individuals, substring properties between string attributes, etc.

nRQL allows for the formulation of conjunctive queries, in which query variables are used to be bound to those ABox individuals that satisfy the query. nRQL queries will make use of arbitrary concept and role names in TBoxes. For example, given a TBox that describes common terminology in a family domain, including concepts such as *Father*, *Mother*, *Child* etc, and their relationships such as *has-child*, the following nRQL query –

```
(retrieve (?X ?Y)
(and
        (?X Mother)
        (?Y Child)
        (?X ?Y has-child)))
```

retrieves all mother and child pairs that are in the has-child relationship. This query first declares two query variables *X* and *Y*, which will be bound to ABox individuals. The constraints *(?X Mother)* and *(?Y Child)* restrict the query variables to be bound to instances of *Mother* and *Child*, respectively. The next constraint *(?X ?Y has-child)* states that a pair of individuals bound to *X* and *Y* has to be in the *has-child* relationship.

The main features of the nRQL language can be summarized as follows (Racer, 2005):

- Complex queries are built from query atoms – nRQL offers concept query atoms, role query atoms, constraint query atoms, and *SAME-AS* query atoms (as well as some auxiliary query atoms). Query atoms are combined to form complex queries with the query constructors such as *and*, *union*, *neg*, and *project-to*.

- nRQL has a well-defined syntax and clean compositional semantics – nRQL only offers so-called must-bind variables which range solely over the individuals of an ABox. Thus, a concept query atom such as *(?X Mother)* has exactly the same semantics as *(concept-instances Mother)*, i.e. X is only bound to an ABox individual if this individual satisfies the constraint – an instance of concept *Mother*. Here, satisfies means that the query resulting from substituting all variables with their bindings is logically entailed by the KB.

- *Negation as failure (NAF)* as well as *true classical negation* is available – NAF is especially useful for measuring the degree of completeness of a modeling of the domain of discourse in the KB (e.g. we can ask for the individuals from the ABox which are *known to be woman but not known to be mothers*).

- Special support is provided for querying the concrete domain part of an ABox. nRQL allows for the specification of complex retrieval conditions on concrete domain attributes fillers of ABox individuals. For example, we can query for the individuals

that are adults, that is, the filler of the *has-age* concrete domain attributes of these individuals of type *real* must satisfy the complex concrete domain predicate ≥ 18.

- A projection operator *project-to* for query bodies – nRQL is more expressive than relational algebra (or non-recursive Datalog (Hall, 1992)); however, the semantics of nRQL is specified in an algebraic way.

- Complex TBox queries are also available – These enable users to search for certain patterns of sub/super relationships between concepts in TBoxes.

For a more complete and detailed coverage of Racer and nRQL, we refer the reader to (Haarslev & Möller, 2003; Racer, 2005).

2.3. Query on Source Code

In order to understand a software system, maintainers often spend considerable time in exploring source code. This includes browsing the code and searching the parts of interest (Lethbridge & Anquetil, 1997). Brachman et al (Brachman, Devanbu, Selfridge, Belanger, & Chen, 1990) have conducted a study in which they observed maintainers of a large software system, measuring the time the maintainers spent performing different categories of maintenance tasks. They found that up to 60% of the time spent on these maintenance tasks was directly related to performing simple searches across the entire software system. A more recent empirical study also shows that, for typical software maintenance tasks, programmers spend approximately 35% of their time with "the

mechanics of redundant but necessary navigations between relevant code fragments" (Ko, Myers, Coblenz, & Aung, 2006)

Furthermore, as the size and complexity of software systems increases, a systematical process of program comprehension is often not possible. Maintainers often adopt therefore an "as-needed" strategy (Soloway, Pinto, Letovsky, Littman, & Lampert, 1988) to search and understand only these source code part that are related to a particular maintenance task. Based on these observations, developing program comprehension tools that support queries on source code becomes an important research area in the program comprehension domain.

Existing query tools can be classified into two main categories based on their underlying models – lexical query tools and syntactical query tools. Lexical query tools support maintainers by searching the source code through keywords or regular expressions, while syntactical query tools support source code searching, by specifying some properties of the syntactical structure.

2.3.1. Lexical Queries on Source Code

Browsing and searching texts by specifying keywords or patterns are facilities provided by almost every text editor and software development environment. In order to be flexible and fast, the majority of these approaches are based on lexical structure, and patterns can only be applied to small predictable chunks of text – mostly to a line of source code. The patterns specified by users are strings of characters, some of which are

intended to be match exactly (e.g. keyword), and others to match one or more of a set or range of characters. The latter ranges in expressiveness from the simplest wildcard pattern language with two match facilities (one-character and many-character match) through full regular expressions with alternation, sequence, grouping, and iteration. (Bull, Trevors, Malton, & Godfrey, 2002).

2.3.1.1. grep Based Tools

The *grep* (Magloire, 2000) tool is a utility that performs regular expression matching over files on a line-by-line basis. Although many program comprehension tools have been proposed, the family of *grep based* tools is still considered the most frequently used ones. In (Singer & Lethbridge, 1997), Singer and Lethbridge considered the success of grep based tools is mainly because their following strengths:

> *The grep tool excels at performing a specific task. Consequently, it is easy to specify a search and the results are returned quickly, frequently with a relevant match. When the search fails, little time or cognitive effort is wasted. In addition, the only required arguments for a grep search are a target pattern and a search domain, all other information is optional. The user does not need to learn regular expression syntax and grep command options in order to use the tool. This knowledge can be acquired as needed, and therefore results typically in a smooth transition from novice to expert.*

In (Sim, 1998), Sim also considered the following additional criteria that contribute to the success of grep based tools:

- Portable and flexible – grep tool is programming language independent. Software maintainers can bring their skills with this tool to any project, in any programming language. While grep is primarily a UNIX tool, implementations are available on other operation systems.

- Little overhead – No indexing of the search domain is needed. Users are not required to generate and index a fact base of the source before using grep.

- Responsiveness – no initial set up for the search is required allowing searches to be performed with out any preprocessing.

However, there exist also a number of limitations of grep based tools. These limitations include among others:

- The interpretation of output requires additional effort – When grep returns a large number of matches, it is difficult for a user to identify the most relevant match. Furthermore, locating the search results in the source code might not be easy, resulting in situations where the results themselves may also need to be searched. In addition, since each match consists of a single line, it often doesn't provide enough contexts to interpret.

- No approximate searches – Matches can not be approximate and must be exact according to regular expression rules. If there is a spelling mistake in a search target, there is no facility in grep to deal with that;

- No semantic searches – The grep utility treats all input as straight text. When searching source code, there is no way to limit the search domain, e.g. identifiers or comments.

- No memory – Search targets or contexts are not stored and cannot be revisited for refinement or modification.

- No browsing – The search results cannot be browsed like hypertext – clicking on a result to display the original place.

The success of grep has led to a family of tools based on grep. One of grep's limitations is that matches must appear on a single line. The *cgrep* (context grep) (Clarke & Cormack, 1996) tool addresses this issue by treating the input as a character stream and interpreting the new-line character as ordinary text, so that it can return matches with arbitrary sizes.

The *agrep* tool (Wu & Manber, 1992) is another extension to the original grep tool, with three major modifications. (1) It allows approximate matches by permitting a user-specified number of substitutions, insertions, or deletions, which could be used in situations when the maintainer didn't know the extract name of an identifier. (2)

Instead of returning a single line, agrep can return matching records, such as entire email messages. (3) It supports logical combination of patterns using AND or OR.

2.3.1.2. LSME

During program comprehension, programmers often need to search the source code not only by specifying keywords or regular expressions, but also by restricting the search to some structural properties of the result. For example, a maintainer might want the result to be an argument of a function, or an if-statement. In order to perform such a query, source code query tools usually parse the source code and thus allow user to perform such structural queries. In contrast, LSME (Lexical Source Model Extraction) technique (Murphy, 1996) is a lexical approach that analyzes source code without using programming language parsers.

LSME provides a lightweight source code analysis approach that permits maintainers 1) to use regular expressions to describe patterns of interest in system artifacts; 2) to specify actions to execute when a pattern is matched to part of an artifact; and 3) optionally, to specify operations for combining matched information to compute structural interactions.

LSME is similar to the UNIX tool awk (Aho, Kernighan, & Weinberger, 1979) in the sense that its searching specifications closely resembles a script or a program rather than a command-line utility. For example, a maintainer who wants to query function calls

between functions from the C source code for gcc might use the patterns shown below as input to the LSME tools.

```
(1)    [ <type> ]  <functionName> \( [ { <formalArg> }+ ] \)
       [ { <type> <argDecl> ; }+ ] \{
(2)          <calledFunctionName> \( [ { <parm> }+ ] \) @
             Write ( functionName, " calls ", calledFunctionName @
```

The first pattern describes how to recognize function definitions in K&R-styled C code, while the second pattern describes the form of calls within a function body. The first pattern states that a function definition might start with an optional type declaration followed by the name of a function, an opening parenthesis, optional formal arguments, a closing parenthesis, optional argument type declarations, and an opening curly brace. Based on these pattern descriptions, a scanner can be generated that will scan a software artifact for these patterns. The generated scanner will attempt to match zero or more call patterns for every occurrence of the function definition pattern. When a call is identified in the source code, the action code (within @ symbols) is executed, writing out the call interaction.

One of the problems with the LSME tools is its accuracy. For example, LSME tools may generate all the function calls in the source code, and some information are false positive (Murphy, 1996). However, the flexibility a maintainer gains in such lightweight lexical tools often outweighs their trade-offs in accuracy.

2.3.1.3. Source Code Search Engine

In recent years several source code search engines were made available online, for example Google Code Search [1], Krugle [2], and Koders [3]. These tools are typically backed by crawlers that collect open source code from the Internet. The collected code is then indexed and stored into a large repository. Users can search the repository by specifying regular expressions, implementation languages, and specific projects etc. The search results are ranked by their relevance or more advanced models, such as PageRank (Brin & Page, 1998).

Source code search engines are essentially lexical approaches that neglect the syntax and structure of software implementation. However, the high-volume data they can handle and their advanced information retrieval models and ranking algorithms provide additional benefits over many existing approaches.

Limitations of Lexical Query Tools

Lexical query tools are successful due to their flexibility and tolerance. These tools are programming language independent, and thus are not sensitive to the source code structure. They are capable of scanning various kinds of software artifacts such as data files or documentation files. These tools are also applicable in situation when a system is

[1] http://www.google.com/codesearch
[2] http://www.krugle.com/
[3] http://www.koders.com/

not compilable (e.g. source code is incomplete or only partially available), such tools are also applicable.

The major limitation of lexical query approaches is the “semantic noise” causing false positive results. Typical semantic noise includes 1) identifiers with longer names that include the specified keyword; 2) comments that include the specified keyword; and 3) identifiers and comments match the specified regular expression but are not desired.

Besides these limitations, source code texts are more structurally complex than what can be captured by lexical approaches. Paul and Prakash (Paul & Prakash, 1994a) have discussed many limitations of regular expression matching, among them are:

- Writing a pattern to distinguish nesting structure of programming language statements is difficult, and sometimes even impossible.
- Writing a pattern to distinguish variable declarations, characterized by order-independent strings of attributes (e.g. `extern long int x`) is difficult and unwieldy.
- Many implementations of regular expression matching do not allow patterns to match across line boundaries.

In summary, lexical approaches lack abilities to capture the structure of source code; they have also difficulties to search keywords or patterns that are related to particular structures. For example, for lexical approaches it is practically impossible to distinguish a keyword “student” in a comment from a variable declaration statement.

2.3.2. Syntactical Queries on Source Code

As mentioned previously, many complex source code queries cannot be addressed by lexical approaches. Among queries that are frequently used by maintainers (Paul & Prakash, 1994b) and cannot be supported by lexical approaches are:

- Queries may be based on global structural information in the source code, e.g. relationships between program entities such as files, functions, variables, types etc.

- Queries can also be based on statement-level structural information, e.g. looking patterns (e.g. loop statements) that fit a programming plan or a cliché (Rich & Waters, 1990).

- Queries may focus on flow information derived by static analysis such as data flow and control flow analysis, e.g. to compute program slices (Weiser, 1984) or to find statements that may affect the value of a particular variable.

- In addition, maintainers also often want to perform queries based on high-level design information of the software, e.g. to query whether the software implements a particular design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) or whether it confirms a reflexion model (Murphy et al., 1995).

For these types of complex queries, syntactic information of source code has to be modeled, and query languages for these underlying models have to be defined. A

number of such source code query systems have been proposed, and many of them share the following common structure:

1) A repository that stores source code information;

2) Tools that populate the repository with structural and/or program flow information, such as language parser, static/dynamic analyzers;

3) An interface for the user to submit queries and obtain results; and

4) A query processor that handles queries by retrieving the repository.

These systems are typically distinguished by the form of underlying source model, and the way of query language defined.

In the remainder of this section, we survey source code query systems that utilize syntactic source code structures and then discuss their strengths and limitations.

2.3.2.1. Relational Approaches

A number of approaches (CIA (Chen, Nishimoto, & Ramamoorthy, 1990), CIA++ (Grass, 1992), grok (Holt, 1996), Relation Partition Algebra(Paul & Prakash, 1994b), sgrep (Bull et al., 2002), etc) have been proposed using relational models to represent source code structures. Among these approaches, CIA (the C Information Abstraction) System (Chen et al., 1990) is one of the earlier source query systems. It extracts relational information from programs written in C, according to a conceptual model and stores the extracted

information in a database. Users of CIA system can then query the information using the relational query language provided by the database system.

As part of the conceptual model for C programs, objects and relationships are defined at a selected abstraction level. The resulting abstraction defines the knowledge available in the database (Chen et al., 1990). Figure 2-2 illustrates this model:

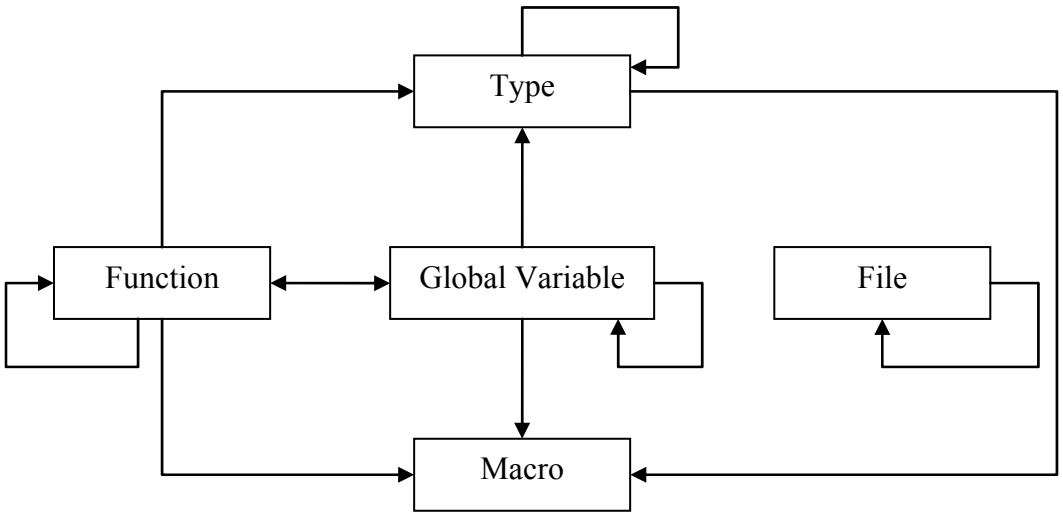

Figure 2-2 Conceptual Model of CIA

In the CIA model, five types of C language objects are defined – *File, Macro, Data type, Global variable and Function*. Each of these objects has a set of attributes. For example, a function has attributes such as 1) file that this function is defined in, 2) data type the function returns, 3) name of the function, 4) whether it is a static function, 5) its start line number and 6) its end line number.

The conceptual model does not explicitly define the order of the attributes and their storage format, leaving these details to the relational schema.

A relationship - *reference* is defined in the conceptual model. If an object A has a reference relationship with object B, then A cannot be compiled and executed without the definition of B. Table 2-1 shows all the meaningful reference relationships among the five kinds of object in C program.

In a next step, a C parser extracts objects and relationships from a program, and stores the information in a database. Users of CIA can query the information through InfoView - a collection of tools specifically designed for access the conceptual model, or through QUEL - a relational query language. Three major types of information retrieval are illustrated – 1) info: retrieval of attribute information of an object; 2) rel: retrieval of relationships between two object domains; and 3) view: view the definition of an object.

Table 2-1 Relationships in CIA Model

Object kinds #1	Object kinds #1	Interpretation
File	File	File1 includes Files2
Function	Function	Function1 refers to Ffunction2
Function	Global Variable	Function refers to Global Variable
Function	Macro	Function refers to Macro
Function	Type	Function refers to Type
Global Variable	Function	Global Variable refers to Function
Global Variable	Global Variable	Global Variable1 refers to Global Variable2
Global Variable	Macro	Global Variable refers to Macro
Global Variable	Type	Global Variable refers to Type
Type	Type	Type1 refers to Type2
Type	Macro	Type refers to Macro

The CIA system is an early implementation of a source code query system capturing only five basic concepts of the program language, and the relationships are also fairly simple. However, the CIA system, as well as its predecessor – OMEGA system (Linton, 1984), provide the foundation for many succeeding systems in the way it uses a relational model to represent source code information and to express queries. CIA++ (Grass, 1992), grok (Holt, 1996), Relation Partition Algebra (Feijs, Krikhaar, & Ommering, 1998), and sgrep (Bull et al., 2002) are all based on such a relational model. They extend the CIA system by 1) supporting a variety of programming languages, 2) providing more fine-grained source code objects, and 3) by including more expressive query languages.

For example, in addition to common source code objects such as files, functions, classes etc, the sgrep tool also supports complex query expressions constructed from operators and sub-expressions, including

Table 2-2 Operators Provided by sgrep Tool

Operations	Computes	Purpose
$\mathbf{R_1} \circ \mathbf{R_2}$	Relation	Relational Composition
id(**s**)	Relation	Identity relation on set **s**
$\mathbf{R}^+$	Relation	Transitive closure of **R**
s.**R**	Set	Project set **s** through relation **R**
R.**s**	Set	Project set **s** through relation **R** backwards
rng(**R**)	Set	Range of relation **R**
dom(**R**)	Set	Domain of relation **R**

2.3.2.2. Source Code Algebra

Source Code Algebra (SCA) (Paul & Prakash, 1994b) is an approach to set up a formal framework for source code queries. In (Paul & Prakash, 1994b), they first argued that relational source code models are inadequate in formally modeling the complex structure of source code. From their perspective, these inadequacies are due to the use of relational algebra – the foundation of relational database and its lack of support for a wide variety of atomic (e.g. *integer*, *string*) and composite data types (e.g. while-statement *has* condition and body, or statement-list is a *sequence* of statement) in source code. Besides, relational models cannot characterize type hierarchy relationships (e.g. for-statement *is kind of* loop-statement).

They address these modeling limitations of relational algebra through their Source Code Algebra (SCA) (Paul & Prakash, 1994b), which is based on the generalized order-sorted algebra (Bruce & Wegner, 1990), as the foundation for building a source code query system. Generalized order-sorted algebra is essentially a many-sorted algebra (in contrast to one-sorted relational algebra) with a partial order defined on its sorts. They consider SCA to be capable of modeling source code structures through sorts that correspond to the various source code data types (atomic and composite) and are ordered by the subtype of relationship.

In SCA, the source code is modeled as:

Definition 1: Let S be a set of sorts. In SCA, S contains all the atomic data types and composite data types. Thus

$S = ATOM \bigcup COMP$, where

$ATOM = \{INTEGER, BOOLEAN, FLOAT, ...\}$

$COMP = \{while - statement, ..., statement, statement - list, ...\}$

Definition 2: A generalized order-sorted algebra A is a 3-tuple $(S, \leq, OP)$, where S is a set of sorts, $\leq$ a partial ordering defined on the sorts, and OP a set of functions (called the operator set) such that:

1) a set A_s, called the *carrier set* or *domain* of s, is defined for each $s \in S$

2) the signature of a function $\sigma \in OP$ is given by $\sigma : A_{s_1} \times A_{s_2} \times ... \times A_{s_n} \rightarrow A_s$, where $s_1, s_2, ..., s_n, s$ are elements of S.

3) if $t \leq s_i$, then elements of A_t can substitute as elements of A_{s_i}

Based on this model, operators can be defined on these sorts to express queries. The operators of SCA can be classified into three categories:

1) Operators defined on atomic data types, such as $+, -, *, /$ arithmetic operators, and, or, not Boolean operators, and $=, <, >, \leq, \geq$ relational operators.

2) Operators defined on individual objects, such as *closure* (computing transitive closure for a given object and a list of attribute names), and *identical* (testing whether two objects are same).

3) Operators defined on collections (i.e. sets and sequences), such as selection, projection reduction, union, subset, set to sequence, head and tail of sequence, concatenation of sequence, subsequence, etc.

Using the SCA operators, source code queries can be expressed as algebraic expression. An evaluation of an algebraic expression on the source representation yields the result of the query. For example, the following expression will find the file that has the maximum number of functions:

$$head_1(order_{no\_of\_func,>}(set\_to\_seq(extend_{no\_of\_func:=size\_of(funcs)}(FILE))))$$

where the FILE objects are extended with a new field, namely no_of_func, and the set of these new objects is then converted into a sequence and arranged in decreasing order of their no_of_func. Finally, the head of this sequence become the file with maximum functions. Views can also be created by algebraic expression, and queries can be performed based the created views.

SCA sufficiently models source code information and contains very expressive operators for supporting queries of interest for software maintainers. However, it is unclear to us whether SCA expressions can be evaluated efficiently. In addition, SCA model only

provides fixed terminologies in describing source code structures, and therefore, lacks of flexibility for specific comprehension tasks.

2.3.2.3. Graph Based Approaches

More recently graph based approaches, like CodeSurfer (Anderson & Zarins, 2005), Xfig (Sartipi & Kontogiannis, 2001), GUPRO (Lange, Sneed, & Winter, 2001), CLG (Kullbach, 2000), etc. , have been introduced to address the limitations of relational approaches (Klint, 2003),

These graph based approaches represent source code in graph structures (node as source code object, and edge as relationship), and use GReQL (Kullbach & Winter, 1999) language to query the graph.

In GUPRO (Lange et al., 2001), source code is parsed into graph structures. The conceptual model of the graph structure is illustrated in Figure 2-3, in which four types of nodes, Module, Class, Method, and Attribute are defined to capture major objects found in source code. Relationships between objects, as well as their restrictions, are also specified.

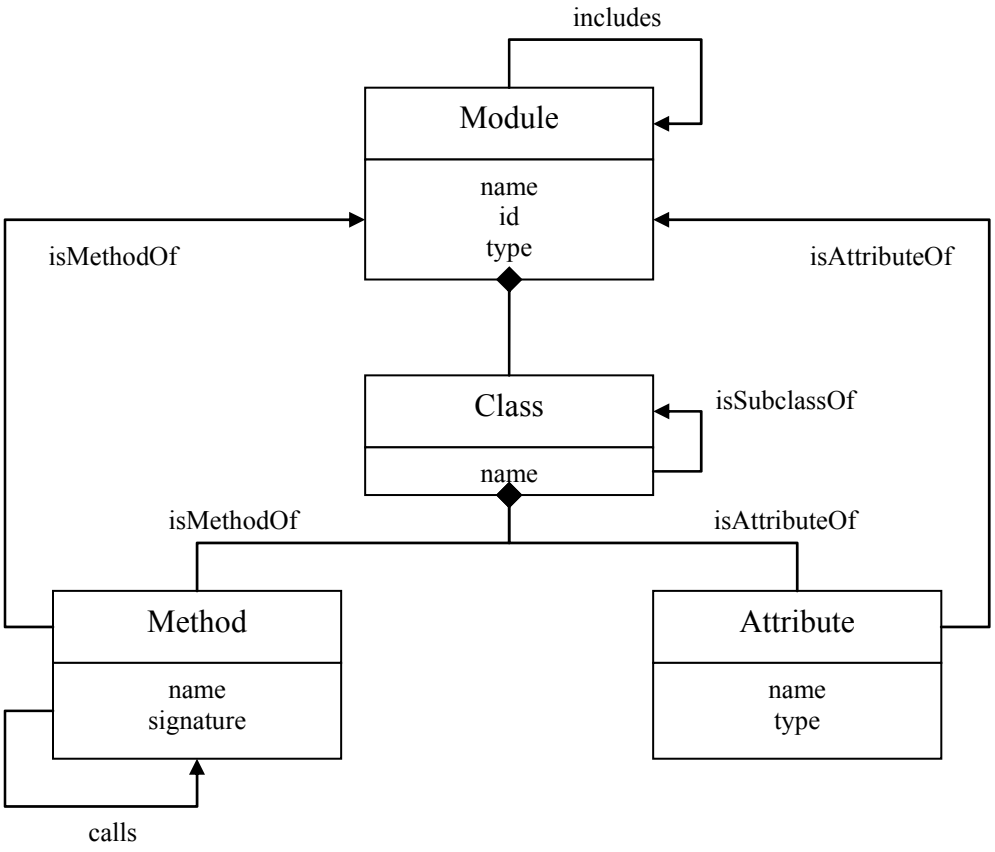

Figure 2-3 Conceptual Model of GUPRO

GUPRO accesses the graph structure using GReQL (Graph Repository Query Language) (Kullbach & Winter, 1999). GReQL is a pure declarative query language that provides a powerful approach for the analysis of entities and their relationships in a graph structure. A typical GReQL query consists of three clauses: FROM, WITH, and REPORT. The FROM clause declares graph objects (nodes and edges). The WITH clause specifies predicates that have to be satisfied by the declared graph objects. GReQL supports first-order predicate logic on finite sets. The path expression can be used to describe regular path structures, e.g. sequences, alternatives and iterations (reflexive and transitive

closure) of paths in a given graph. The REPORT clause displays the query results in the form of table. GReQL queries can be nested in REPORT part.(Kullbach & Winter, 1999)

For example, the following query reports the name of each class and the name the most general super class of that class:

```
FROM c, super : V{Class}
WITH c --> {isSubclassOf}* super
AND
outDegree {isSubclassOf} (super = 0)
REPORT c.name, super.name
END
```

The FROM part declares two variables: c and super, of type Class. The WITH part restricts the possible assignments such that c has to be a subclass of super on arbitrary level and that super must not be a subclass of any other class. The REPORT part specifies to report the name of class c and the name of its highest super class.

Limitations of Syntactical Query

Representing relational views of source code and using SQL-styled language to query them have some inherited problems: 1) Using relational models it is difficult to capture complex source code structures, especially type hierarchies; 2) the expressiveness of SQL is very restricted in its ability to express transitive closure (Klint, 2003); and 3) there will be considerable performance drawback when extending a relational query language.

Graph based query languages overcome some of the problems of relational approaches. The graph query language GReQL (Kullbach & Winter, 1999) supports transitive closures and enables the user to formulate regular path expressions. The major benefit of GReQL is the ability to directly traverse the graph, whereas in SQL different tables have to be joined many times. However, current graph based approaches are mostly memory based, and graph query languages typically cannot capture type hierarchies. In addition, the semantics of the underlying graph model and query languages of these approaches is often defined only optionally. As a result, depending on the semantic used to model the system, different graph based models may behave differently for the same system.

Some other approaches use network structure to represent source code. For examples, in Rigi (Muller & Klashinsky, 1998), a special purpose semantic network data model is adopted to represent objects and relationships in source code. A language (RCL) is designed to manipulate them. However, the problem with these approaches is that they lack query languages with well-defined semantics, and the queries themselves have to be procedural.

Both lexical query tools and syntactical query tools are difficult to explore domain semantics often captured and conveyed by source code. For example, in order to find a variable that represents “current temperature”, maintainers have to use regular expressions provided by lexical tools or LIKE clause in SQL to find clues, which is opportunistic and inaccurate.

2.4. Ontology and Its Applications in Software Engineering

Software development is known to be a knowledge-intensive activity. Selfridge (Selfridge, 1992) states that “it is the amount and scope of relevant knowledge that makes software so difficult”. Knowledge-Based Software Engineering (KBSE) addresses this issue by using formally represented knowledge and inference services to support various activities of software development (Devanbu & Jones, 1994).

As size and complexity of software systems increases, KBSE tools are challenged with limitations caused by their expressive power and reasoning efficiency. i.e. the more expressive power of the representation formalisms, the less reasoning efficiency, and vice versa. DLs provide us “an effective tradeoff” (Devanbu & Jones, 1994) between these two. KBSE tools may benefit from DLs by 1) formal semantics based on first order logic, 2) proven expressive power in representing domain knowledge, 3) thoughtfully understood computational complexity, and 4) efficient DL system support.

In this section, we focus on the applications of DL knowledge bases in software engineering.

2.4.1. LaSSIE

LaSSIE (Premkumar, Ronald, Peter, & Bruce, 1991) is the first DL application of software information system (SIS) (Brachman et al., 1990). It addresses the *invisibility* (F. P. Brooks, 1987) problem of program understanding, i.e., the structure of software system is hidden to the programmer/ maintainer, by providing a large knowledge base and a

natural language interface. Users of LaSSIE may query about the operations of a PBX switches system using natural language, and LaSSIE can recognize instances of such operations implemented in the software.

The knowledge base in LaSSIE consists of a source code ontology (source code model) and application domain ontology – telephony (domain model). Based on the two models, the KB system may analyze the query from users (using domain concepts) and reason, mainly based on subsumption and rule triggers, the presence of domain concepts in source code. LaSSIE is built on a frame-based (one of the ancestors of DL) knowledge based system – KANDOR.

It has been shown in (Premkumar, Ronald, Peter, & Bruce, 1991) that the LaSSIE system can cut down on the time maintainers spent in discovery source code. However, since the mapping between source code object and domain concepts is manually created, which is time-consuming. The maintenance of the mapping is also difficult because the domain model changes over time and new features are added to the source code (Baader, 2003). Therefore, such approach is not feasible for large software systems or for time-pressured maintenance tasks.

2.4.2. CODE-BASE

CODE-BASE (Selfridge, 1991; Selfridge & Heineman, 1994) system is a complementary approach to the LaSSIE system, by providing additional efficiency and ease-of-use features. The CODE-BASE system also automatically extracts a large amount of facts

from the source code and stores these facts in a repository. A heuristic algorithm is adopted for providing these data in a "on demand" manner. CODE-BASE provides users also with the ability to define new concepts and include these new concepts into rules, and thus further extend the LaSSIE knowledge base. (Selfridge & Heineman, 1994)

CODE-BASE supports a number of graphical tools for viewing and browsing the information in the knowledge base. While this is less significant from the general perspective of DLs, it is important from the standpoint of developing knowledge-based systems. (Baader, 2003)

CODE-BASE is built on the **CLASSIC** (Patel-Schneider, McGuiness, Brachman, Resnick, & Borgida, 1991) Knowledge Representation System.

2.4.3. KITSS

KITSS (Nonenmann & Eddy, 1992) is a knowledge based system that can provide assistance in converting semi-formal test case specification into executable test script. After a natural language parser transforms test case specification into a mid-format, a **CLASSIC** knowledge base is involved to resolve references in noun phrases – classify individuals into their corresponding domain concepts. For example, a sentence "Station A calls station B" is recognized as 1) A is an individual of *Station*, 2) B is another individual of *Station*, 3) B is the filler of role *calls*. KITSS mainly uses the classification/subsumption mechanism of the **CLASSIC** system to perform such jobs.

The reference resolution part of the knowledge base in KITSS contains a telephony domain model and a set of rules that retrieve individuals from test case specifications.

2.4.4. CBMS

A serial of works (Welty, 1995a; Welty 1995b; Welty 1996; Welty 1997) from Chris Welty, so called Code-Based Management Systems (CBMS), is another attempt to apply DLs to software comprehension. CBMS provides a comprehensive fine-grained representation of source code, which is basically based on abstract syntax tree. Such representation can then be used to navigate source code and to detect side-effects.

In addition to some traditional concept reasoning services in DL systems, CBMS provides two reasoning services on roles, namely, role inverses and path tracing (role composition) (Welty, 1997). Each role in CBMS has its inverse, e.g. a role that describes an assignment statement *changes* a variable has an inverse role – a variable is *changedBy* an assignment statement. A role in CBMS can also be defined by the composition of other roles, e.g. a role *changedInFunction* is defined as *changedBy* $\circ$ *implementationOf*, and thus links a variable with a function where the variable is changed.

Welty considered the most compelling result from the use of CBMS to be the automatic detection of side effects. For example, one kind of side effect detected is that assignment statements modify global variables. In order to detect such side effect, a concept *AssigmentSideEffect* is defined as equivalent to *GlobalAssignmentStatement*,

and a forward chaining rule – $AssignmentSideEffect \Rightarrow DirectSideEffect$ is added into the knowledge base to classify all assignments that change global variables as assignment side effect and direct side effect. From a program comprehension point of view, such types of classifications make it possible to localize information that otherwise would be difficult (or at least time consuming) to discover. Identifying these side effect inferences are automatically provided and do not require any additional effort from the developer or maintainer.

CBMS is built on the **CLASSIC** Knowledge Representation System.

2.4.5. Other Works

In addition to above KBSE tools, other researchers have proposed to apply DL in the software engineering domain. In (Welty & Ferrucci, 1999), Welty et al presented a formal ontology that may capture and re-use architectural level knowledge from software documents. Möller, in (Möller, 1996), presented an approach that integrates OO programming methodologies and DL knowledge system. In (Yang, Cui, & O'Brien, 1999) Yang et al presented an approach that can extract domain ontologies from legacy systems written in COBOL for program comprehension and re-engineering. In (Berardi, Calvanese, & Giacomo, 2003), Berardi et al showed that UML diagrams can be formalized using DLs, and they conducted a set of experiments to check the consistency of UML diagrams using modern DL reasoners.

With the adoption of DL as a major foundation of the recently introduced Semantic Web and Web Ontology Language (OWL) (McGuinness & Harmelen, 2004), there is a trend to utilize ontologies or introduce taxonomies as conceptual modeling techniques into the software engineering domain. However, these existing approaches mainly focus on knowledge representation and sharing. Therefore, automated reasoning and formal aspects of the domains are less important for these applications.

For example, in requirement engineering, ontologies have be used to support requirements management (Decker, Rech, Ras, Klein, & Hoecht, 2005), traceability (Mayank, Kositsyna, & Austin, 2004), and use case management (Wouters, Deridder, & Paesschen, 2000).

In software maintenance domain, Ankolekar et al (Ankolekar, Sycara, Herbsleb, Kraut, & Welty, 2006) provide an ontology to model software, developers, and bugs. Ontologies have also been used to describe the functionality of components using a knowledge representation formalism that allows more convenient and powerful querying. One approach implementing this is the KOntoR (Happel, Korthaus, Seedorf, & Tomczyk, 2006) system that allows storing semantic descriptions of components in a knowledge base and performing semantic queries on it. In (Jin & Cordy, 2003), Jin et al. discuss an ontological approach for service sharing among program comprehension tools.

Chapter 3. Ontology-based Program Comprehension Model

In this chapter, we present a novel ontology-based program comprehension approach. The research is motivated by the shortcomings of existing approaches that support searching and relating information in software artifacts. In particular, we provide ontological representations for both source code and software documents. These representations are based on DLs and allow programmers to reason about transitive relations and concept hierarchies in software artifacts, which are typically not supported by existing approaches. The ontological representations also support the definitions of new vocabularies (concepts and/or relationships) related to a programmer's specific comprehension tasks. Therefore, our approach provides programmers with an extensible conceptual model and a query language, which are more flexible than existing approaches. Furthermore, utilizing existing techniques such as source code analysis and text mining, our approach supports information integration from both source code and software documents at finer granularity levels.

In Section 3.1, we establish the relationship between ontologies and program comprehension. Section 3.2 presents the research hypothesis and research goals, as well as a set of criteria to validate our hypothesis. Section 3.3 describes in detail the software ontology – the conceptual model of software artifacts. An ontology-based program comprehension methodology is introduced in section 3.4, followed by several

concrete application examples of our approach in section 3.5. Our major research contributions are summarized in section 3.6.

3.1. The Role of Ontology in Program Comprehension

In order to understand a program, programmers spend considerable time in searching and relating information in existing software artifacts. A recent empirical study (Ko et al., 2006) shows that for typical software maintenance tasks, programmers spend approximately 35 percent of their time with "the mechanics of redundant but necessary navigations between relevant code fragments". They concluded that program comprehension is essentially a process of searching, relating, and collecting information of perceived relevance. Based on this observation, we believe that in order to facilitate programmers in searching and relating behaviors during program comprehension, a successful approach must address two fundamental questions: (1) *what* types of information are programmers searching for? 2) *How* are these pieces of information related? Since the result of a program comprehension process is a programmer's mental representation of the software system (Walenstein, 2002), we also believe that the answers to these two questions is directly related to the study of (3) what constitute a human mental model?

As suggested by Corritore and Wiedenbeck (Corritore & Wiedenbeck, 1999), a programmer's mental model of a software program consists of relationships between code elements and the purpose and intent of these elements. Research in cognitive science further points out that a mental model may take many forms, but its content

normally constitutes an *ontology* (Johnson-Laird, 1983). In the context of program comprehension, such an ontology consists of concepts and their relationships in the domain of programming techniques, data structure and algorithm, design techniques, and specific application domains etc, as well as a set of code elements corresponding to the instances of these concepts.

Based on these mental model theories, we consider the information that programmers search for is a set of concepts and/or concept instances related to their specific tasks, and these concepts and/or concept instances are related according to their prior programming and application domain knowledge. The program comprehension process therefore can be described as an iterative process of concept recognition (i.e. *searching*) and relationship discoveries (i.e. *relating*) in software artifacts (Figure 3-1).

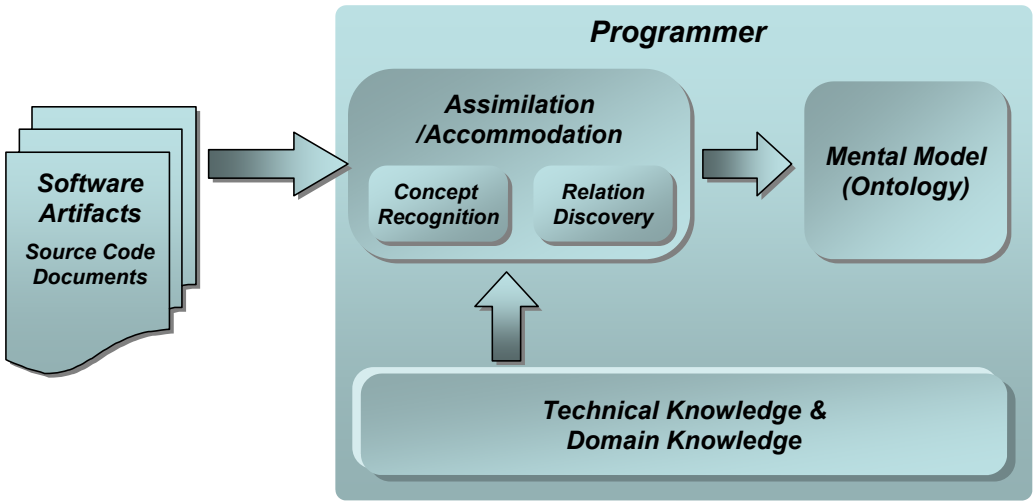

Figure 3-1 The Role of Ontology in Program Comprehension

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 their relationships are identified, and new concepts may also be 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.

Concept Recognition (searching) – One can consider the following example, in which a maintainer might start with a simple ontology about the Java language. Such an ontology might include concepts like *method*, *variable* and *modifier.* During code/documentation review, the maintainer tends to first recognize instances of these existing concepts, due to their familiarity. In situations where further analysis is required, for example to study the accessibility of source code entities, programmers would use additional concepts, such as *public method* or *private variable*, to analyze the source code.

In some cases, new concepts may be learned from source code or different types of documents. For example, a design document may state that a class is a *façade class*. For a maintainer who is not familiar with such a concept, further consultation of the documents or analysis of the source code would be required, to understand that the façade class is the public access point of a component. As a result, the newly learned concepts will be used to enrich the ontology.

In addition, for many program comprehension tasks, maintainers also have to understand the application domain functionalities implemented by the source code. This requires maintainers to map domain concepts to the identified source code entities. For example, a class may be recognized as representing a domain concept, like bank draft, or a method may be mapped to interest calculation of a bank account.

Relationship Discovery (relating) – A typical comprehension process also includes activities of discovering properties of identified entities, i.e. their relationships with other entities. Simple relationships, such as a variable is *defined in* a class or a method *calls* another method, can be recognized instantly. More implicit relationships can be constructed from these simple relationships. For example, a class C_1 *uses* another class C_2 if either a variable defined in C_1 has the type C_2 or a method defined in C_1 accesses methods or fields defined in C_2.

Result – The result of a program comprehension process is an ontology that captures both, relevant concepts and relationships required for a particular comprehension task, as well as a set of source code entities and their relationships corresponding to an instance of that ontology. The ontology can therefore be considered as the content of maintainer's current mental model of a program. Such an ontology also forms the basis for the next iterations of the program comprehension process.

As ontologies play an important role in the process of understanding a program, we call this model an ***ontology-based program comprehension model***. In contrast with other program comprehension models, which mainly focus on the *strategies* that

programmers may adopt (e.g. top-down (R. Brooks, 1983) (Soloway & Ehrlich, 1984), bottom-up (Shneiderman & Mayer, 1979) (Shneiderman, 1980), or integrated strategy (Letovsky, 1986) (Von Mayrhauser & Vans, 1995)), our model focuses on the *knowledge* required to understand a program. This model therefore is neutral with regard to these strategy-based models, and in combination with these models, it can be used to develop a more comprehensive *ontology-based program comprehension methodology*, which describes both the information required for comprehension tasks and the strategies to obtain this information.

3.2. Research Hypothesis and Research Goals

3.2.1. Research Hypothesis

In this research we present a new software comprehension methodology that utilizes ontologies and automated reasoning techniques. In particular, we focus on representing software artifacts, such as source code and documents, in the form of formal ontologies, and study how DL reasoners can assist programmers in searching and relating information in these artifacts. Our research hypothesis can be defined as follows:

> Research Hypothesis:
>
> *Ontological representations of software artifacts can potentially provide additional benefits over existing approaches in facilitating programmers searching and relating information in software systems.*

We expect our research hypothesis to hold if the following acceptance criteria can be validated:

Cognitive Support

Cognitive support, in the context of program comprehension, refers to the leverage of innate human abilities, such as visual information processing, to improve human understanding and cognition of complex software systems (Walenstein, 2002). Stelzner and Williams (Stelzner & Williams, 1988) further point out that the essence of cognitive support is try to establish the mapping between the user's mental model of a system and the system model itself. A majority of program comprehension approaches address cognitive issue by representing software systems in various forms, such as visualization tools or code beautifier. One shortcoming of these approaches is that their efforts are only focused on *representing* the system, rather than *constructing* the mapping. We believe that an ontological model of software systems is not only another representation form, but also provides programmers with additional means of constructing knowledge. This assumption is based on the following observations.

In addition to the mental model theory (Johnson-Laird, 1983), which suggests that the content of a mental model constitutes an ontology, research in cognitive development or constructive learning also regards concepts as building blocks of human constructive learning (Novak, 1998; Piaget, 1971). In this theory, humans actively learn new facts based on prior knowledge and by grouping its possibilities in four different operations: *conjunction*, *disjunction*, *negation*, and *implication*. The learning process consists of two

complementary parts: *assimilation* where the new facts are incorporated without changing the prior knowledge, and *accommodation* where the prior knowledge has to accommodate itself to the new facts and thus these new facts lead to a reorganization of the existing knowledge.

Both the mental model theory and the cognitive-development theory have been previously applied to program comprehension (Corritore & Wiedenbeck, 1999; Rajlich & Wilde, 2002). As stated earlier, a comprehension process can be considered as a learning process where, through code/document reading, programmers adopt (by assimilation or accommodation) new facts based on their current mental model (Rajlich & Wilde, 2002). The result of a comprehension process is a programmer's mental model that consists of relationships between code elements and the purpose and intent of these elements (Corritore & Wiedenbeck, 1999).

An important observation is that programmers often assimilate or accommodate new facts using logical operations such as conjunction, disjunction, or negation of prior knowledge (Piaget, 1971). These logical operations typically have direct correspondences in formal ontology/DLs, in which the semantics for concept subsumption, conjunction, disjunction, and negation etc are all well defined. Therefore, comparing to existing software representations, we expect that our ontological representations for software systems can provide additional benefit to assist programmers in concept/relationship constructions, i.e. *mapping* software system to their mental models.

Source Code Model

As discussed in chapter 2, using relational or other ad-hoc models to capture the structure of source code is difficult mainly due to the fact that these models are inadequate in representing the type hierarchy (subsumption) of programming language (Paul & Prakash, 1994a), and relational query languages such as SQL only have restricted support for transitive closures (Klint, 2003).

By utilizing very expressive DLs, we expect our ontological model to overcome some of these limitations. In particular, the ontological model used in our approach supports concept hierarchies, role hierarchies, transitive roles, and concrete domain concepts (e.g. line number of a source code element). The reasoning complexity of $\mathcal{ALCQHI}_{R+}(\mathcal{D})$ logic is decidable, and there exists optimized ontology reasoners, such as Racer (Haarslev & Möller, 2003), FaCT++(Tsarkov & Horrocks, 2006), Pellet (Sirin et al., 2006).

Query Language

The query languages provided by state-of-the-art DL systems are often more expressive than relational algebra (or non-recursive Datalog) (Haarslev et al., 2004). For example, the nRQL query language provided by the Racer system allows for the formulation of conjunctive queries, in which query variables are used to be bound to individuals that satisfy given restrictions. Users can therefore compose complex queries to traverse relational paths by specifying relationships between query variables. Moreover, in addition to the Open World Assumption of DLs, nRQL also supports localized Negation

As Failure (NAF) semantics, which is especially useful for programmers to reason the *absence* of information in a model. Therefore, in companion with the source code model, we also expect that our approach is able to provide a powerful query language to assist programmers in navigating and reasoning about the ontology.

Information Integration

One of the major challenges for programmers while performing a comprehension task is the need to integrate information from a multitude of often disconnected artifacts. These artifacts are created as part of the software development process, and include, among others, source code, software documents (e.g., requirements, design documentation). From a programmer's perspective, it becomes essential to establish and maintain the semantic connections among these artifacts. However, major obstacles exist in integrating information from both software documents and source code. These obstacles, among others, include (1) these software artifacts are written typically in different languages (natural language vs. programming language); (2) they describe a software system at different abstraction levels (design vs. implementation); and (3) they are created initially during forward engineering and often not well maintained as part of the maintenance effort.

Traditional approaches dealing with software documents are mainly based on Information Retrieval (IR) techniques (Baeza-Yates & Ribeiro-Neto, 1999), which address the indexing, classifying, and retrieving of information in natural language documents.

In the software engineering domain, IR techniques are mostly applied for indexing software documents and automatically linking software requirements and implementation artifacts. Antoniol et al (Antoniol, Canfora, Casazza, & De Lucia, 2000) addressed traceability problems by linking software documents and code using both probabilistic methods and a Bayesian classifier. Marcus et al (Marcus & Maletic, 2002; Marcus, Sergeyev, Rajlich, & Maletic, 2004) used Latent Semantic Indexing (LSI) (Landauer, Foltz, & Laham, 1998), 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. Hayes et al (J. H. Hayes, Dekhtyar, & Osborne, 2003) also demonstrated a tool that uses a vector space model to improve requirements tracing. 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 (Kowalski, 1997) and ontology-based models (Khan & McLeod, 2000; Khan, McLeod, & Hovy, 2004) 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.

Ontologies have been commonly regarded as a standard technique in representing domain semantics and resolving semantic ambiguities. Therefore, introducing an ontological representation for software artifacts will allow us to utilize existing techniques, such as Text Mining (TM) or Information Extraction (IE) (Feldman & Sanger, 2006), to "understand" parts of the semantics conveyed by these informal information resources, and thus to integrate information from different sources at finer granularity levels.

3.2.2. Research Goal

The general goal of this research is to develop an ontology-based program comprehension approach to support programmers searching and relating information in software systems. As DLs have been long regarded as standard ontology languages, our research goal therefore can be specified as:

> Research Goal:
>
> *Develop a Description Logic Knowledge Base to support the ontology-based program comprehension model.*

This research goal can be further decomposed into several sub-goals, they are:

- Develop a software ontology that is capable of representing various artifacts in software systems, such as source code and software documents.

- Specify a methodology that guides programmer using the ontology to accomplish common comprehension tasks.

- Implement tools to automatically populate the ontology based on existing software artifacts.

- Design a query language that allows programmers navigating and querying over the populated ontology to facilitate their searching and relating behaviors.

- Provide case studies and application examples to study the applicability and limitations of our approach.

3.3. A Software Ontology

Source code and software documentation are primary artifacts that are used during program comprehension. These artifacts typically contain knowledge that is rich in both structural and semantic information. A uniform ontological representation for source code and documentation enables the use of knowledge conveyed by these artifacts to provide support for programmers in constructing appropriate mental models for software systems. In this section, we present a software ontology for representing knowledge in software systems. This ontology consists of two sub-ontologies: a source code ontology and a documentation ontology. The details of these two sub-ontologies, as well as specific design considerations are discussed in this section.

3.3.1. Source Code Ontology

Source code is an implementation artifact that plays a dominant role in program comprehension, as in many cases, it is the only available artifact for programmers to perform maintenance tasks.

Within our approach, a source code ontology has been developed to formally specify major concepts (e.g. Class, Method, Variable, etc.) and their relationships (e.g. their containing relation, referential relations, etc.) found in Object-Oriented Programming languages. In our implementation, this ontology is further extended with additional concepts and roles needed for specific programming language(s). For example, we included Java specific concepts, like Interface and Package etc, according to the Java Language Specification (Joy, Steele, Gosling, & Bracha, 2000).

Concepts

The source code ontology contains more than 100 common concepts in the programming language domain, and some of them are Java specific. This ontology is derived from parse trees; however, it contains rich semantic associations between source code entities that are obtained from source code analysis, such as indirect method calls, variable read/write, etc. Figure 3-2 shows part of the taxonomy of the source code ontology.

As suggested in (Schneiderman, 1986), 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.

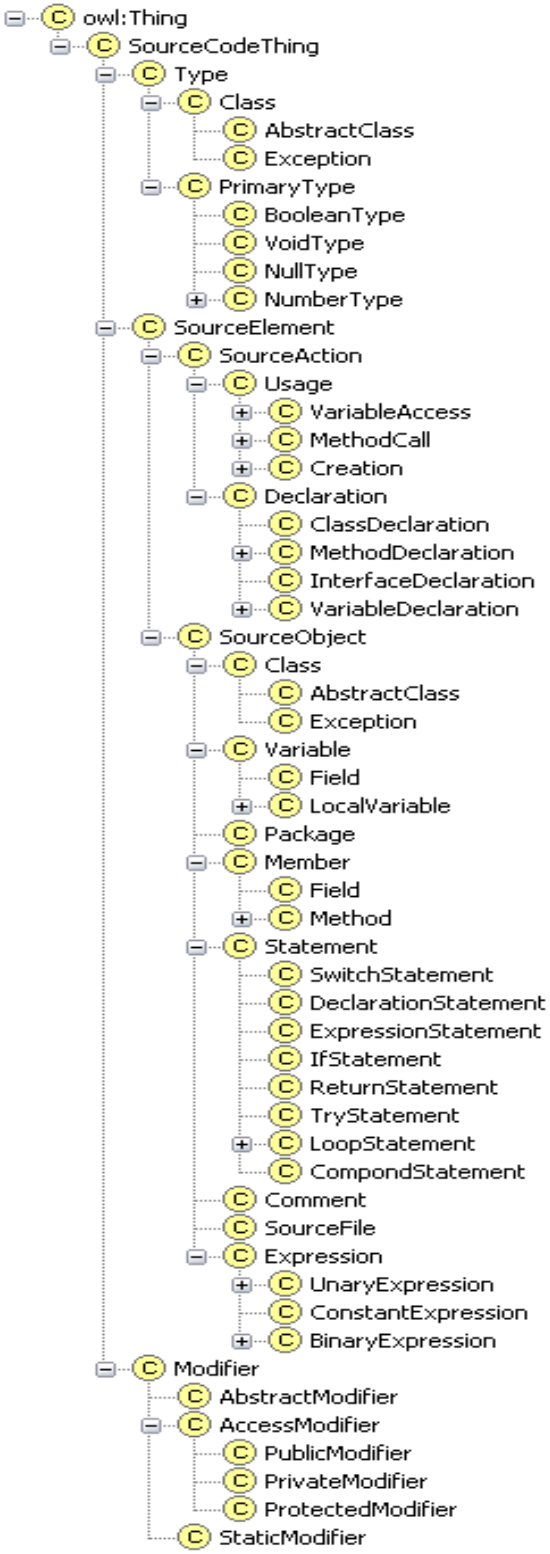

Figure 3-2 Concept Hierarchy in the Source Code Ontology

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 has direct correspondences in the source code, and therefore, instances of these concepts can be automatically identified by source code parsing.

A *SourceAction* typically denotes a semantic association between two or more *SourceObject*s. For example, A *MethodCall* action refers to: within the body of a method, there is a method invocation expression 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 etc.

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 source code entities. For example, the concept *Type* represents the abstract type of source code entities, concept *Modifier* is used to specify the accessibilities of entities, such as public, private, etc.

The use of DL allows us to formally characterize the *subsumption* relationship between concepts. A concept *C* is considered as a sub-concept of *D* if all instances of *C* are also

instances of *D*. Therefore, for example, 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 –

$$Method \sqsubseteq SourceObject \sqcap \exists definedIn.Class$$

specifies that a method is a type of source object, and it has to be defined in a class, and

$$Variable \sqsubseteq \forall hasType.Type$$

means that a variable must have a type.

Some concepts are defined by providing *necessary and sufficient conditions*. For example –

$$Member \equiv Method \sqcup Field$$

defines that a class member is either a method or a field (and vice versa), and

$$PublicField \equiv Field \sqcap \exists\ hasModifier.PublicModifier$$

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

Relationships

Within the source code ontology many roles are defined to specify relationships between individuals. A partial view of the role names and their descriptions is shown in Table 3-1.

Table 3-1 Role Names in the 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

Furthermore, we further provide for each of these roles also their corresponding *inverse*. 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 method *M write* a variable *V*, then *V* is *writtenBy M*.

One of the advantages of using DLs is the 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 $(a, b)\in R$ holds, then $(a, b)\in S$ 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 $(M, V)\in read$ is discovered, then $(M, V)\in access$ is also implied.

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

$$classReadVariable \equiv contain \circ read$$

In such a case, if class C contains a method M, and the method M reads a variable V, that is, $(C, M)\in contain$, and $(M, V)\in read$, then $(C, V)\in classReadVariable$ 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: $(aMethod, aField) \in write$.

Characterizing Object-Oriented Programming

We design our ontology to characterize three key features of 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.

2) *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 language, which can only be captured partially by static

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 same signature, i.e. they both override the same method defined in the superclass. If two methods are in *sameSignature* relationship, either of them may potentially be invoked when the method in superclass is invoked, e.g. through dynamic binding (polymorphism).

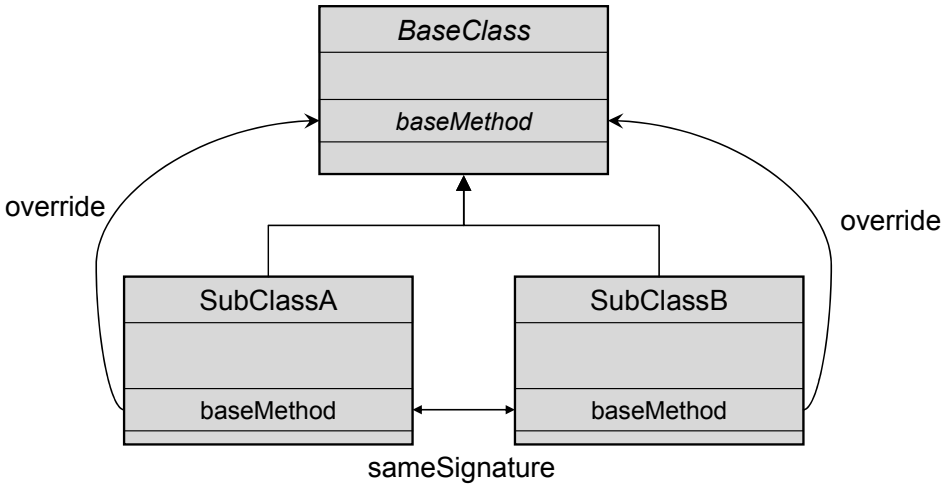

Figure 3-3 Characterizing Polymorphism Mechanism of OOP

For example, as shown in the UML class diagram in Figure 3-3, both SubClassA and SubClassB inherit from BaseClass. Therefore, the following relations can be automatically identified:

$$(SubClassA.baseMethod, BaseClass.baseMethod) \in override$$

$$(SubClassB.baseMethod, BaseClass.baseMethod) \in override$$

and

(SubClassA.baseMethod, SubClassB.baseMethod) $\in$ *sameSignature*

Both the *override* and the *sameSignature* role are transitive. The *sameSignature* role is also symmetric, and is a sub-role of *override*.

3.3.2. Documentation Ontology

The documentation ontology consists of a large body of concepts that are expected to be discovered in software documents. At the current stage of our research, this ontology is mainly based on concepts found in technical domains, such as programming languages, algorithms, data structures, and design decisions especially design patterns and software architectures. Figure 3-4 illustrates the top level concepts of the documentation ontology.

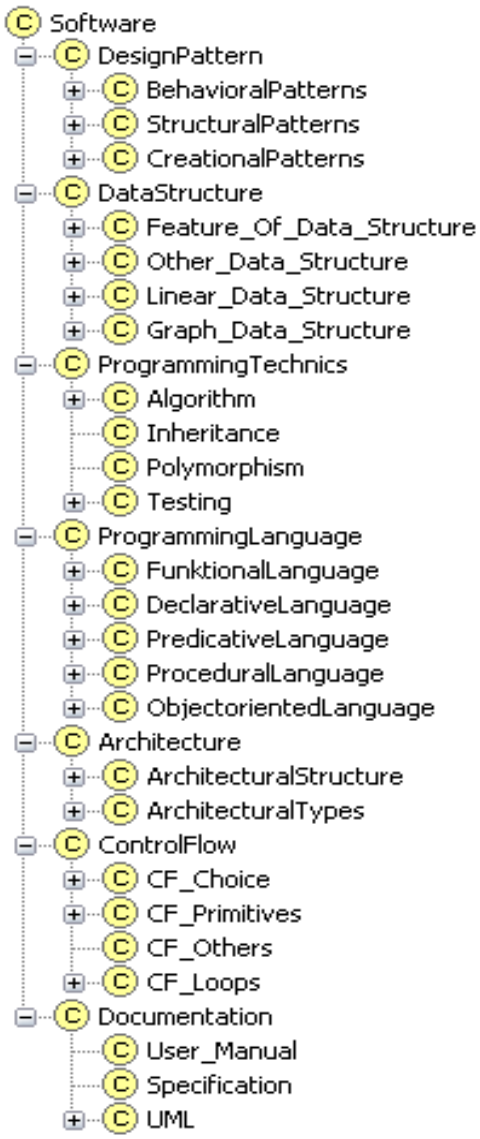

Figure 3-4 Top-level Concept Hierarchy in Documentation Ontology

The software documentation ontology has been designed to support automatic population through a text mining system by adapting the ontology design requirements outlined in (Witte, Kappler, & Baker, 2006) for the software engineering domain. Specifically, we included:

A *Text Model* to represent the structure of documents, e.g., classes for sentences, paragraphs, and text positions, as well as NLP-related concepts that are discovered during the analysis process, like noun phrases (NPs) and coreference chains. These are required for anchoring detected entities (populated instances) in their originating documents.

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

Relationships between the classes, which include the ones modeled in the source code ontology. These allow us to automatically restrict NLP-detected relations to semantically valid ones (e.g., a 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, as we describe in more detail in chapter 4.

3.4. A Knowledge Base

3.4.1. Software Knowledge Base

Our software ontology is a formal conceptual model of typical software artifacts. Utilizing a state-of-the-art ontology reasoner, such as Racer, we can provide programmers with a Software Knowledge Base to assist them reason about various properties of software systems, as shown in the following diagram –

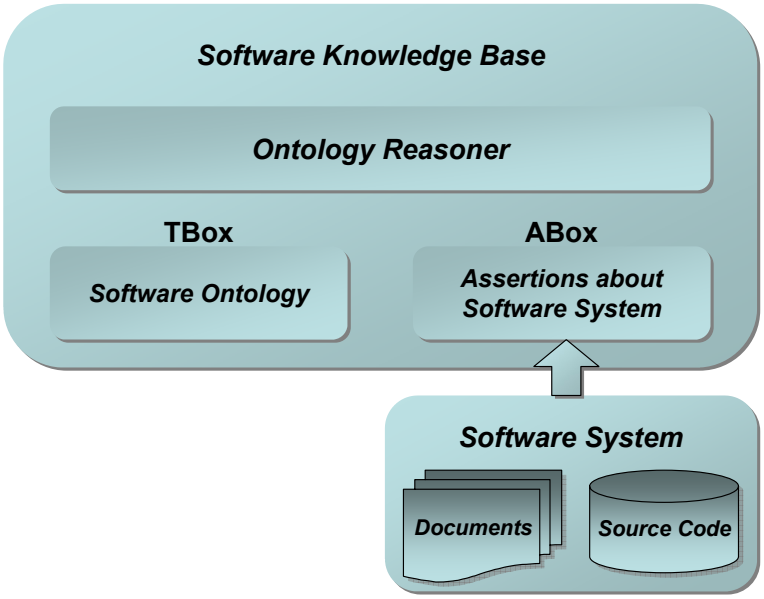

Figure 3-5 Software Knowledge Base

Within this knowledge base, the software ontology contains terminologies used to describe software systems, i.e. a TBox. Assertions about a software system can be extracted from software artifacts, including documents and source code. These assertions thus constitute an ABox for the software ontology and a particular software

system. The ontology reasoner provides inference services based on the ontology and these assertions.

Typical ontology reasoners provide TBox inference services such as concept consistency, subsumption, satisfiability, and TBox consistency, classification etc, as described in Section 2.2.4. However, in this research, as we aim to assist programmers to comprehend a particular concrete software system, we focus more on ABox inference services. Concrete examples to demonstrate the use of these inference services are presented in Section 3.6.

3.4.2. Query Interface

The ontology reasoner Racer provides a very expressive ABox query language – nRQL to assist users to retrieve concept and role instances in an ontology. An nRQL query uses concept and role names defined in the ontology to specify properties of the result. In a query itself, variables are used to be bound to Abox instances that satisfy given restrictions, the so-called Active Domain Semantics (Haarslev, Möller, & Wessel, 2005).

However, the use of the nRQL language is still largely restricted to programmers with a good mathematical/logical background. An nRQL query, although comparatively straightforward, is still difficult for programmers to understand and even more difficult to create. Furthermore, the declarative nature of the nRQL language restricts users with the capability of manipulating the query result. For example, in many cases, users of our knowledge base need to count the number of returned instances, or use regular

expression to filter out some of the result. We have developed a scriptable query language based on JavaScript to bridge the gap between practitioners and the formal representation of the knowledge base. We provide a set of built-in functions and classes in the JavaScript interpreter – Rhino* to allow users to compose scripts for querying the ontology.

Ontology Object

A built-in object called "*ontology*" is provided to represent the software ontology. This object is the main interface for users to interact with the knowledge base. Functions provided by the *ontology* object include –

Table 3-2 Functions of Built-in Object *ontology*

add_concept(*name*)
Add an atomic concept into the ontology.
add_concept(*name*, *description*)
Add a concept into the ontology by providing its description.
add_role(*name*)
Add a role into the ontology.
add_role(*name*, *inverse*)
Add a role into the ontology and specify its inverse role.

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

add_role(*name*, *transitive*, *inverse*)
Add a role into the ontology and specify its inverse role and whether it is transitive.
implies($role_1$, $role_2$)
Specify $role_1$ as a sub-role of $role_2$.
add_instance(*name*)
Add an instance into the ontology.
add_instance(*name*, *concept*)
Add an instance of a specific concept into the ontology.
add_relation($instance_1$, *role*, $instance_2$)
Specify $instance_1$ and $instance_2$ have a specific relationship (*role*)
retrieve_instance(*concept-description*)
Retrieve all instances of a concept or concept description
retrieve_instance(*concept-description*, *regular-expression*)
Retrieve instances of a concept or concept description that match the specified regular expression
retrieve_relation(*role*)
Retrieve specific type of relationships
retrieve_relation(*domain*, *role*, *range*)
Retrieve specific type of relationships (*role*), whose domain is *domain*, and range is *range*.
define_query(*name*, *Query*)
Define a named query to the ontology
query(*Query*)

Perform a query by giving a *Query* object

Query and Result

We provide two built-in classes, *Query* and *Result*, to assist users in composing queries and manipulating the results. Table 3-3 lists the major functions provided by the *Query* class.

Table 3-3 Methods of Built-in Class *Query*

declare(var_1, var_2, ….)
Declares query variables in the query, which will be bound to instances that satisfy the restriction.
restrict(*var*, *C*)
Specifies a concept restriction for the query result – the query variable *var* has to be an instance of *C*, which can be either a concept name or concept description.
restrict(*var*, *R*, *object*)
Specifies a relation restriction for the query result – the query variable *var* has an *R* relation with the *object*. The *object* can be either a variable or a named individual.
known_relation(*var*, *R*, *object*)
Specifies a relation restriction – the query variable *var* has at least one explicitly modeled *R* relation with the *object*. The *object* can be either a variable or a named individual. This restriction allows localized Negation-As-Failure semantics to be applied to the query.
not_concept(*var*, *C*)

Specifies a concept restriction – the query variable *var* must NOT be an instance of *C*, which can be either a concept name or concept description.
no_relation(*var*, *R*, *object*)
Specifies a relation restriction – the query variable *var* has no *R* relation with the *object*. The *object* can be either a variable or a named individual.
no_known_relation(*var*, *R*, *C*)
Specifies a relation restriction – the query variable *var* has NO explicit modeled *R* relation with individual whose type is *C*, which can be either a concept name or concept description This restriction allows localized Negation-As-Failure semantics to be applied to the query.
retrieve(var_1, var_2, ….)
Specifies the result that will only include instances bound to specified query variables.

Users can use these existing concept and role names in the ontology to retrieve instances with specified properties. The typical steps involved in composing a query are as follows:

(1) declare query variables;

(2) specify restrictions to these query variables using concept names, concept descriptions, role names, and/or instances in the ontology; and

(3) submit the query to the built-in object *ontology*, and obtain the result.

For example, the following script (Query 3-1) can be used to retrieve all public methods in an ABox –

```
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);
```

Query 3-1 JavaScript Query – Public Method

Query 3-1 first declares two variables M and P, and then specifies restrictions that M shall be bound to an instance of *Method*, and P to an instance of *PublicModifier*. The third restriction specifies individuals that match for M and P shall be in a *hasModifier* relationship. The next statement states that this query result is a projection that only contains instances bound to M.

Query 3-1 is equivalent to the following nRQL query –

```
(RETRIEVE (?M)
    (AND (?M Method)
        (?P PublicModifier)
        (?M ?P hasModifier)))
```

Query 3-2 nRQL Query – Public Method

The result of a query is a set of tuples that satisfy the specified restrictions. The class *Result* is used to represent this query result. This class provides the following methods –

Table 3-4 Methods of Built-in Class *Result*

get(*var*)
Return a list of instances that have been bound to *var*.
get(*var*, *position*)
Return an instance that is bound to *var*, and at the specific returned *position*.
get(*var*, *regular-expression*)
Return a list of instances that have been bound to *var*, and satisfy the given *regular-expression*
size()
Return the size of the returned tuples.

Users of our knowledge base can further manipulate the result by using methods of the *Result* class. For example, for the result returned by Query 3-1, the following script retrieves all instances that match regular expression "sort*", i.e. all public methods whose name start with "sort" –

result.get("m", "sort");*

Logic Functions

In addition to built-in classes and objects, we also provide a set of built-in logic functions to facilitate the composition of concept descriptions. Some of the built-in logic functions

are summarized in Table 3-5, in which C and R are string parameters that denote concept names and role names respectively.

Table 3-5 Built-in Logic Functions

AND (C_1, C_2, …)	conjunction of C_1, C_2, …
OR(C_1, C_2, …)	disjunction of C_1, C_2, …
NOT(*C*)	negation of *C*
EXIST(*R*, *C*)	concept that there exists a relation *R* whose filler is type of *C*
ALL(*R*, *C*)	concept that all its *R* fillers have to be instance of *C*
KNOWN(R)	concept that has explicitly model relation *R* within KB

Using these built-in functions and terminology provided in the software ontology, the concept public method can be specified as:

ontology. add_concept("PublicMethod",
AND("Method", EXIST("hasModifier", "PublicModifier")));

This is equivalent to the following DL concept description –

$PublicMethod \equiv Method \sqcap \exists\ hasModifier.PublicModifier$

The new concept *PublicMethod* thus enriches the existing vocabulary of the ontology. Users can refer to this new concept later in their specific queries. For example, Query 3-1 can now be simplified as follows -

var result = ontology.retrieve_instance("PublicMethod");

Query 3-3 Simplified Public Method Query

3.5. An Ontology Based Comprehension Methodology

In Section 3.1, we have characterized program comprehension as an iterative process of concept recognition and relationship discovery. The result of each iteration corresponds to an ontology that includes relevant concepts and relationships required for a particular comprehension task, as well as a set of source code entities and their relationships corresponding to an instance of the ontology. This ontology-based program comprehension model focuses on the *knowledge* required for understanding a software program. Therefore, by adapting appropriate knowledge acquisition strategies, such a model can support program comprehension in either *bottom-up*, *top-down, integrated*, or *as-needed* approaches.

In what follows we introduce a novel ***ontology-based program comprehension methodology*** that describes information required for program comprehension, as well as strategies to obtain this information. This methodology can be used to guide programmers performing program comprehension tasks through ontology exploration and automated reasoning. Figure 3-6 illustrates basic activities involved in the ontology-based program comprehension methodology. The following subsections describe when and how these activities may be invoked in the context of specific comprehension strategies.

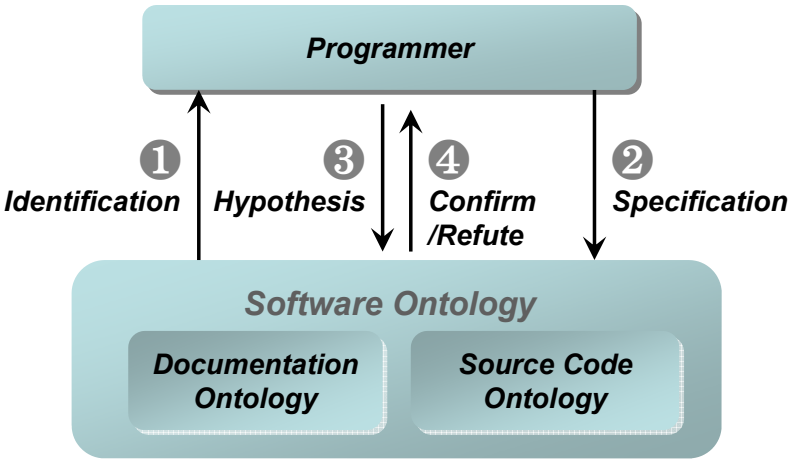

Figure 3-6 Ontology-based Program Comprehension Methodology

3.5.1. Bottom-up Comprehension Strategy

For the bottom-up comprehension strategy, programmers understand a program based on *chunking* (Letovsky, 1986). Chunks are portions of code that the programmer recognizes. Larger chunks contain smaller chunks nested within them. The programmer pieces together his/her understanding by combining chunks into increasingly larger chunks. In the perspective of ontology-based comprehension, chunks are concept instances. Smaller chunks or larger chunks denote concept instances at different granularity levels, such as statements, method, class, or package levels. Therefore, recognizing smaller chunks, and formulating larger chunks are a set of activities called *concept identification and specification*. More specifically,

❶ Concept identification refers to an activity by which programmers retrieve information related to a specific portion of a software artifact. The software ontology contains rich concept instances at different granularity levels, as well as the

relationships among these instances. The ontology therefore can facilitate programmers in identifying concepts and relationships that are related to a specific portion of the code, e.g. a method he/she is currently reading. For example, during the reading of a smaller chunk – a variable usage, a programmer may query the ontology about the type of this variable, or the place where this variable is defined.

In order to understand a larger chunk, such as a class, a programmer may first retrieve smaller chunks that are contained within the larger chunk, such as methods defined in the class, or variables used in the class. Only once all retrieved smaller chunks have been comprehended, the programmer is able to obtain a complete understanding of the larger chunk.

❷ After such a localized understanding of a program chunk has been obtained, programmers often continue the comprehension process by introducing additional concepts or relationships with regards to their specific maintenance tasks, or specifying understood chunks as instances of existing concepts/relationships. The newly added knowledge then becomes an integrated part of the ontology and can be reused in other comprehension tasks. For example, in order to comprehend data structures implemented in a software system, a programmer may define new concepts such as *Array*, *LinkedList*, *Stack*, or *Heap* etc as part of the ontology. He/she thus can use these new concepts to specify recognized chunks, e.g. a class is an instance of *LinkedList*. This information can then be further reused by the programmer to identify larger chunks. For example, a *Stack* is implemented by a *LinkedList*.

Figure 3-7 illustrates a typical bottom-up comprehension process as supported by our ontology based program comprehension methodology –

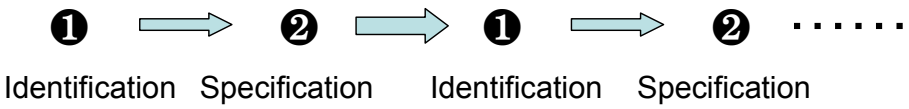

Figure 3-7 Bottom-up Comprehension Process

3.5.2. Top-down Comprehension Strategy

In situations where a top-down comprehension strategy is adopted, programmers start code reading with certain hypotheses and then confirm or reject these hypothesis based on identified evidences, the so-called *beacons* (F. P. Brooks, 1987). Within the ontology-based program comprehension model, a hypothesis can be regarded as a concept description or a query. When the concept description or the query returns one or more concrete instances, the hypothesis is confirmed and retained, becoming part of their understanding. Otherwise, the hypothesis is discarded.

❸ Programmers typically start the comprehension process by formulating hypotheses concerning properties of the program. The ontology language can help programmers to specify their hypotheses. For example, a programmer may suspect that a class implements a specific design pattern (Gamma et al., 1995). In a next step, an ontology query is composed and performed to validate if any of the software documents describe this class in terms of design patterns. Another example is that a programmer may hypothesize a software package is a self-contained component, i.e. there exists no

dependencies to other classes/packages outside the package it is implemented. A simple query can help the programmer to verify this hypothesis by retrieving all classes that are used by this package, but not defined within it.

❹ The software ontology can answer queries through automated reasoning, and thus confirm/refute these hypotheses. Through each confirmation or refutation, the programmer obtains a better understanding of the program. In the above examples, the documentation ontology was used to identify whether any design patterns are related to the class; and the source code ontology returned the number of classes that are used by the package. These provided answers can help programmers verify their hypothesis, and thus accelerate the comprehension process.

A typical top-down comprehension process is illustrated in Figure 3-8 –

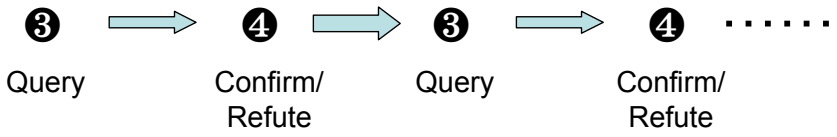

Figure 3-8 Top-down Comprehension Process

3.5.3. Integrated Comprehension Strategy

An integrated comprehension strategy considers that program understanding is built at several levels of abstraction by switching between top-down or bottom-up comprehension strategies (Von Mayrhauser & Vans, 1994). Within our ontology-based

comprehension methodology, the integrated comprehension process consists of a series of activities involving searching (concept identification or ontology querying) and relating (concept specification or query confirmation/refutation). These activities may be invoked at any time during the comprehension process. Due to the fact that the ontology acts as a single information source throughout the comprehension process, activities related to the integrated strategy can be performed in any order, as shown in Figure 3-9.

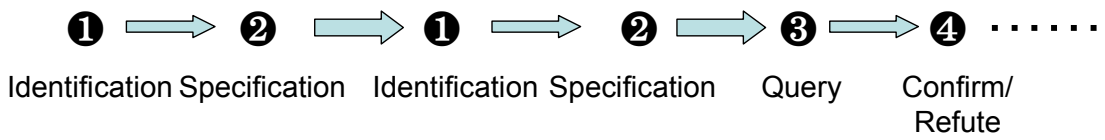

Figure 3-9 Integrated Comprehension Process

3.5.4. As-needed comprehension Strategy

As the size and complexity of a software system increases, a complete understanding of the whole system is less feasible (Rajlich & Wilde, 2002). Instead, programmers often adopt an *as-need* strategy (Koenemann & Robertson, 1991). For this reason programmers tend only to comprehend these portions of the program that are relevant for a specific maintenance task.

Adopting an as-needed strategy as part of the ontology-based program comprehension approach requires a programmer to start the comprehension process with an initial mental ontology. This mental ontology consists of a set of concepts/relationships that

are considered essential for a maintenance task. Through source code and documents readings, instances of the initial ontology are comprehended in the first step (i.e. ontology populating). In the case more concepts/relationships are necessary, the ontology has to be extended, and more effort is required to comprehend instances of these newly introduced concepts/relationships. Therefore, the as-need comprehension process can also be regarded as an iterative process of concept recognition and relationship discovery, except that the comprehension goal is not to construct a complete mental model, but rather a minimum subset that is required for the completion of a particular comprehension task.

Within our methodology, the as-need comprehension strategy is similar to the integrated strategy. The software ontology contains detailed information with regard to source code and documents. Programmers can identify certain subset of the ontology as their starting point. By exploring and querying the ontology, they can not only investigate specific properties of a concept/concept instance, but also enlarge the subset by utilizing existing or newly defined concepts/relationships. The process finishes once they are confident that their understanding is sufficient to complete the particular task.

3.6. Applications

In this section, we present several application examples to demonstrate the applicability and flexibility of the ontology-based program comprehension approach. These

application examples are organized based on the abstraction level of typical maintenance tasks.

Maintainers tend to comprehend first the dependencies between different source code entities, such as classes, methods, and variables (Section 3.6.1). The result of such a dependency analysis forms the basis for analyzing the impact of a change request (Section 3.6.2). Some maintenance tasks require identifying very specific patterns in source code, such as security patterns, which are discussed in Section 3.6.3. We also demonstrate how our approach can be applied to maintenance tasks at higher levels of abstractions, such as Design Pattern or Software Architecture analysis – both are essential analysis techniques for programmers to understand the overall structures of a software system (Section 3.6.4). In the case that software documents are available as an information source, our approach also supports the integration of information found in both source code and documents (Section 3.6.5).

3.6.1. Dependency Analysis

Identifying dependencies between source code entities is one of the most important comprehension tasks towards revealing the structure of a software system. A dependency refers in this context to the semantic association between two source code entities. For example, a class inherits from another class, a method calls another method, or a variable is accessed by a method, etc. Our source code ontology already contains a large amount of fine-grained dependency information about the source code, such as method invocations, variable declarations, and class inheritances, etc. Such

information can be further used to analyze higher level dependency relationships that might exist among source code entities.

In our first application example, the motivation is to identify the *use* dependency between classes. Being able to identify this dependency type is the foundation for many program comprehension tasks, like impact analysis and/or architectural analysis. A *use* dependency at class level (e.g. a class C_1 uses anther class C_2) implies many fine grained dependencies, such as –

1) a variable defined in C_1 whose type is C_2;

2) a method defined in C_1 calls a method defined in C_2;

3) a method defined in C_1 has read or write access to a field defined in C_2;

4) C_1 has a super class which is C_2.

In order to specify the *use* relationship within our source code ontology, a new role – *use* is defined to represent the *use* relation. In addition, this role also has its corresponding inverse role – *usedBy*. (Query 3-4)

```
ontology.add_role("use", true, "usedBy");
```

Query 3-4 Define *use* Role

In order to represent the specialized *use* relationship mentioned before, the following four roles are defined –

ontology.add_role("variable_use");
ontology.implies("variable_use", "use");

ontology.add_role("method_use");
ontology.implies("method_use", "use");

ontology.add_role("field_use");
ontology.implies("field_use", "use");

ontology.add_role("inheritance_use");
ontology.implies("inheritance_use", "use");

Query 3-5 Define Specialized *use* Roles

This query defines four distinctive roles in the ontology, namely – *variable_use*, *method_use*, *field_use*, *inheritance_use*, and specifies them as sub-roles of the *use* role. Query 3-4 and Query 3-5 together specify the structure of the use relationship at object class level, which is also a representation of programmers' knowledge, as shown in the following figure.

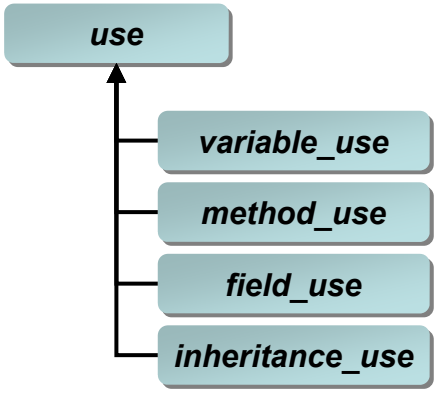

Figure 3-10 Class Level *use* Relationship

Based on the newly defined roles, several queries can then be performed, to create the *use* relationship between classes. For example, the following query (Query 3-6) states that a class C_1 has a *use* relation with another class C_2 if a variable defined in C_1 is of type C_2, i.e. *variable_use* relationship.

```
var variableUse = new Query();
variableUse.declare("C1", "C2", "V");
variableUse.restrict("C1", "Class");
variableUse.restrict("C2", "Class");
variableUse.restrict("V", "Variable");
variableUse.restrict("C2", "typeOf", "V");
variableUse.restrict("V", "definedIn", "C1");
variableUse.retrieve("C1", "C2");
var result = ontology.query(variableUse);
```

Query 3-6 *variable_use* Relationship

This script first declares three query variables – C_1 and C_2 that should be bound to classes, and V that should be bound to variable. These variable declarations are followed then by restrictions specifying that C_2 is the type of V, and V is defined in C_1. The result of this query therefore is a set of class pairs that have the *variable_use* relationship.

Several types of DL reasoning services are involved in answering the above query:

1) ***Concept Subsumption*** – in this query, V is declared as an instance of concept *Variable*. Because in the source code ontology, *Field*, *LocalVariable* and *Parameter* are all sub-

concepts of *Variable*, our DL reasoner will automatically bind instances of *Field*, *LocalVariable* and *Parameter* to V.

2) ***Transitive Closure*** – in the source code ontology, role *definedIn* is a transitive role. Since the query states that V is defined in C_1, the DL reasoner will automatically consider all variables within the class, including variables that are directly defined in the class (i.e. fields), and variables that are indirectly defined in the class (i.e. local variables or parameters).

3) ***Role Composition*** – in this query, C_2 is the type of V, and V is defined in C_1. The relationships between these query variables can be illustrated by the following figure.

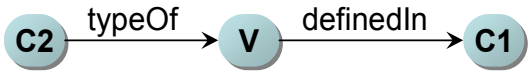

Figure 3-11 Relationships between C_1, C_2, and V

Therefore, the relationship between class C_2 and class C_1 is the composition of two roles *typeOf* and *definedIn*. The role composition reasoning is especially useful for traversing the relation path within a network structure.

4) ***Inverse Role*** – in our source code ontology, the *typeOf* role is an inverse role of *hasType*. Our source code analyzer only identifies *hasType* relationship between variables and types. Therefore, if *(var, type)* $\in$ *hasType* is identified, the DL reasoner can automatically infer the inverse relation *(type, var)* $\in$ *typeOf*.

The result of Query 3-6 is a special type of *use* relation. Users of our knowledge base can explicitly specify the resulting class pairs (C_1 and C_2) having *variable_use* relationships into the ontology, as shown in the following query (Query 3-7) –

```
for(var i = 0; i < result.size(); i++){
   var class_name1 = result.get("C1", i);
   var class_name2 = result.get("C2", i);
   ontology.add_relation(class_name1, "variable_use", class_name2);
}
```

Query 3-7 Specify *variable_use* Relationship into Ontology

In addition to Query 3-6, three more queries are applied to analyze other types of *use* relation between classes –

1) if a method defined in C_1 calls a method defined in C_2, then C_1 and C_2 have a *method_use* relation (Query 3-8);

2) if a method defined in C_1 access a field defined in C_2, then C_1 and C_2 have a *field_use* relation , (Query 3-9); or

3) if C_1 has a super class that is C_2, then C_1 and C_2 have a *inheritance_use* relation (Query 3-10).

```
var methodUse = new Query();
methodUse.declare("C1", "C2", "M1", "M2");
methodUse.restrict("C1", "Class");
methodUse.restrict("C2", "Class");
methodUse.restrict("M1", "Method");
methodUse.restrict("M2", "Method");
```

```
methodUse.restrict("M1", "definedIn", "C1");
methodUse.restrict("M2", "definedIn", "C2");
methodUse.restrict("M1", "call", "M2");
methodUse.retrieve("C1", "C2");
```

Query 3-8 *method_use* Relationship

```
var fieldUse = new Query();
fieldUse.declare("C1", "C2", "M", "F");
fieldUse.restrict("C1", "Class");
fieldUse.restrict("C2", "Class");
fieldUse.restrict("M", "Method");
fieldUse.restrict("F", "Field");
fieldUse.restrict("M", "definedIn", "C1");
fieldUse.restrict("F", "definedIn", "C2");
fieldUse.restrict("M", "access", "F");
fieldUse.retrieve("C1", "C2");
```

Query 3-9 *field_use* Relationship

```
var inheritanceUse = new Query();
inheritanceUse.declare("C1", "C2");
inheritanceUse.restrict("C1", "Class");
inheritanceUse.restrict("C2", "Class");
inheritanceUse.restrict("C1", "hasSuperClass", "C2");
inheritanceUse.retrieve("C1", "C2");
```

Query 3-10 *inheritance_use* Relationship

It has to be noted that the above four special *use* roles can be directly defined using more expressive DL constructors, such as role composition, which is supported by DLs like $ALCFI_{reg}$, (Baader, 2003) or *SROIQ* (I. Horrocks, Kutz, & Sattler, 2006) –

$variable\_use \equiv contains \circ hasType$

$method\_use \equiv contains \circ calls \circ definedIn$

$field\_use \equiv contains \circ access \circ definedIn$

$inheritance\_use \equiv hasSuper$

In such cases, instead of querying and explicitly specify these relationships, the ontology reasoner can automatically infer them according to these role descriptions.

Existing approaches supporting dependency analysis are typically implemented within Integrated Development Environments (IDEs). For example, the Eclipse development environment allows users to query for the declaration and usages (references) of the current variable, method, or class. However, these tools only support limited number of predefined queries. Relational approaches (see Section 2.3.2) supporting source code queries allow the definition of views for each type of dependency. However, tables in these models have to join many times and transitive relations are typically not supported. Recent approaches (Bull et al., 2002) support transitive closure and role composition. However, our approach provides additional benefits by supporting role hierarchies, allowing users to either refer to specific roles individually, or to the role hierarchy in general.

The result of a dependency analysis at the class level is that each class is connected with its dependent classes by a set of *use* relations. These identified dependencies enrich the ontology and then can be further reused by other analysis tasks such as change impact or architecture analysis. The change impact analysis is demonstrated in the following section, while a more detailed architecture analysis case study is presented in Chapter V.

3.6.2. Change Impact Analysis

Many software maintenance tasks require reasoning about the impact of a change before the implementation of the change (Bohner & Arnold, 1996). Impact analysis not only provides programmers with additional insights on the effects of changes on a software system, it also allows programmers to predict the effort required to perform these changes.

The determination of change impact is one of the central problems in software maintenance. Research in change impact analysis involves, among others, the process of identify change impact, requirement change analysis, dependency analysis, traceability between source code and document, impact representation techniques, etc. Traditional impact analysis techniques at source code level (Arnold & Bohner, 2003) typically rely on dependency graphs, on which transitive closures, relation paths, or program slices can be computed.

The ontology-based program comprehension approach not only supports transitive closures and relation paths but also provides a flexible query language that supports impact analysis at different abstraction levels. Furthermore, programmers are not limited by the pre-defined vocabulary in the ontology – they can also create new ones to support customized queries for specific analysis tasks.

In the following example, we assume that in the software system to be comprehended, a specific interface, e.g. *Event* has to be changed. The goal of the analysis is to identify

the impact related to this interface change. In order to identify and retrieve all classes that have used the *Event* interface, we first created the following query that is based on the previously (Query 3-4) introduced new transitive role *use*, –

```
var impactOfEvent = new Query();
impactOfEvent.declare("C");
impactOfEvent.restrict("C", "Class");
impactOfEvent.restrict("C", "use", "org.infoglue.cms.entities.workflow.Event");
impactOfEvent.retrieve("C");
var result = ontology.query(impactOfEvent);
out.println(result);
```

Query 3-11 Change Impact of Event Interface

The result of this query (Query 3-11) is the transitive closure of the *use* relationship between classes. This result provides a general guidance for the impact related to the interface change.

As mentioned earlier, one of the advantages of the ontology-based approach is that programmers can also define their own concepts for specific analysis tasks. For example, the following query (Query 3-12) can be used as a template for class-level impact analysis, in which two new concepts *ClassChanged* and *ClassAffected* are defined.

```
ontology.define_concept("ClassChanged");
ontology.define_concept("ClassAffected",
                        AND("Class", EXIST("use", "ClassChanged")));

ontology.add_instance("class1", "ClassChanged");
ontology.add_instance("class2", "ClassChanged");
```

……

var result = ontology.retrieve_instance("ClassAffected");

Query 3-12 A General Purpose Impact Analysis Query

In this query, the concept *ClassAffected* is defined by providing its concept description –

AND("Class", EXIST("indirectUse", "ClassChanged"))

In this context, the classes being affected are *all classes that use the changed classes*. Within our knowledge base, the reasoner can then automatically infer the instances of *ClassAffected* according to its definition, which is a typical classification service provided by Racer.

The query also specifies a set of classes that will be changed as part of the maintenance task (i.e. $class_1$, $class_2$ …). These classes have to be replaced by users with the class names specific to a particular analysis tasks. The result of the query includes all classes that are either directly or indirectly use these specified classes.

In addition to the above class-level impact analysis, the source code ontology also provides low-level semantic information of source code, which can be used to analyze the change impact at finer granularity levels. This allows for a further reduction of the search space during impact analysis. For example, we assume that within the *Event* interface, just one method – *getEventId()* will be changed. The impact of this change therefore only includes all methods that have directly or indirectly called the

getEventId(). The following query (Query 3-13) can be used to retrieve all method calls to the *getEventId()* –

```
var impactOfEventId = new Query();
impactOfEventId.declare("M", "C");
impactOfEventId.restrict("M", "Method");
impactOfEventId.restrict("C", "Class");
impactOfEventId.restrict("M", "definedIn", "C");
impactOfEventId.restrict("M", "indirectCall",
                                    "org.infoglue.cms.entities.workflow.Event.getEntityId()");
impactOfEventId.retrieve("C");

var result = ontology.query(impactOfEventId);
out.println(result);
```

Query 3-13 Change Impact of getEventId() method

The *indirectCall* in the above script has been defined as a transitive role in our ontology. The result of this query shows the classes that may directly/indirectly be effected by the method *getEventId()*. The computed transitive closure is therefore a subset of the classes retrieved by the previous Query 3-11, and therefore narrows our search space related to the impact of the change.

DL systems, such as Racer, also support the traversal of relation paths within the dependency graph by specifying relationships between query variables. For example, in order to broaden the analysis, we modified now the previous Query 3-13 to retrieve all Java packages that are affected by the method change, i.e. to query the ontology at a higher level of abstraction. This is done by adding another variable P – representing a

Java package, and a new restriction stating that the class C is defined in P. Part of this query is shown below –

```
impactOfEventId.declare("M", "C", “P”);
impactOfEventId.restrict("M", "Method");
impactOfEventId.restrict("C", "Class");
impactOfEventId.restrict(“P”, “Package”);
impactOfEventId.restrict("M", "definedIn", "C");
impactOfEventId.restrict(“C”, “definedIn”, “P”);
impactOfEventId.restrict("M", "indirectCall",
                                    "org.infoglue.cms.entities.workflow.Event.getEntityId()");
impactOfEventId.retrieve("P");
```

Query 3-14 Packages Affected by getEventId() Method Change

Figure 3-12 illustrates the relation path traversed by Query 3-14 to identify all the packages that are affected by the getEventId() method change.

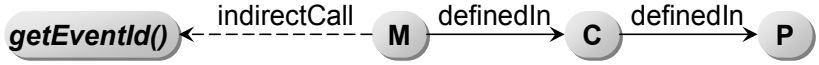

Figure 3-12 Relation Path of Query 3-14

3.6.3. Design Pattern Recovery

Instead of studying the dependencies among source code entities, another common way of program comprehension is to identify patterns implemented within a software system. A pattern describes a commonly recurring structure of components communication that solves a general problem within a particular context (Gamma et al., 1995). The descriptions of patterns contain not only the knowledge about the

components and their communications, but also the alternatives and design decisions associated with particular problems. Recovering patterns existing in a software system may therefore provide programmers with additional insights about the implementation, and thus considerably improve the efficiency of program comprehension (Beck & Johnson, 1994; Keller et al., 1999). Types of patterns in a software system vary from low-level language idioms, Design Patterns(Gamma et al., 1995), to higher level Architecture Styles (Shaw & David, 1996). In this section we demonstrate how our ontology-based program comprehension approach can facilitate programmers to search Design Patterns implemented in software systems.

Existing Design Pattern recovery approaches (Antoniol et al., 1998; Keller et al., 2001) typically depend on predefined design pattern libraries, which limit their applicability to specific program comprehension tasks or specific patterns. In addition, many source code structures might not strictly follow these pattern specifications, but they still convey certain design intensions that may help programmers to understand these structures. However, existing approaches typically lack the ability in supporting users to *relax* or *tighten* the constraints of a pattern specification “on the fly”.

Within our approach, users can specify their own patterns and study these designs and design decisions by adjusting the pattern specifications by increasing or decreasing restrictions of queries.

In this example, we identify the Singleton pattern (Gamma et al., 1995), which is a simple pattern that ensures a class has only one instance at run-time, and provides

public access points to this instance. A typical implementation of the Singleton pattern is a class in which all the constructors have to be private and there is a static public method whose return type is the class itself. The following UML class diagram shows the structure of a singleton class.

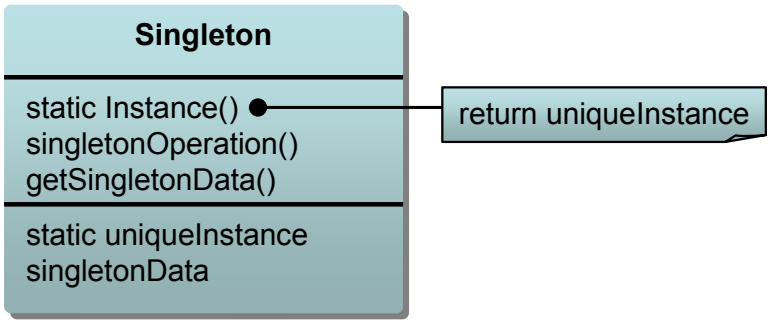

Figure 3-13 Singleton Pattern

Users interested in searching for the existence of Singleton pattern in the source code can use the following very general query:

```
var singleton = new Query();
singleton.declare("C", "I");
singleton.restrict("C", "Class");
singleton.restrict("I", "Constructor");
singleton.restrict("I", "hasModifier", "private");
singleton.restrict("C", "defineConstructor", "I");
singleton.retrieve("C");
var result =ontology.query(singleton);
```

Query 3-15 A General Singleton Pattern Query

The result of this query is a super set of singleton classes, since classes that contain a private constructor, but do not have a static access method are also retrieved. Users

may then further refine the query by adding one more query variable “M” and applying the following additional restrictions –

```
var singleton = new Query();
singleton.declare("C", "I", “M”);
singleton.restrict("C", "Class");
singleton.restrict("I", "Constructor");
singleton.restrict("M", "PublicMethod");
singleton.restrict("I", "hasModifier", "private");
singleton.restrict("C", "defineConstructor", "I");
singleton.restrict("M", "hasModifier", "static");
singleton.restrict("M", "hasReturnType", "C");
singleton.restrict("M", "definedIn", "C");
singleton.retrieve("C");
var result =ontology.query(singleton);
```

Query 3-16 A Refined Singleton Pattern Query

The refined query returns now all classes that have a private constructor and a static method whose return type is the class itself. Similar restrictions can be applied to specify that a singleton class has to define a static field whose type is the class itself. Users may also discover additional design information by comparing the differences between these two queries. Classes retrieved only by the less restrictive query may indicate special design intentions (e.g. private constructor just for internal object cloning) or implementation defects.

It has to be pointed out that in the given example these two queries are only user defined approximations of a Singleton pattern specification. However, the results still

provide significant clues for the search of the Singleton pattern. The following query, using Negation As Failure (NAF) semantics provide in Racer, will identify all Singleton pattern implementations in the program.

```
var singleton = new Query();
singleton.declare("C");
singleton.restrict("C", "Class");
//class C has at least on constructor
singleton.restrict("C", EXIST("hasConstructor"));
// class C does NOT have explicit modeled relationship between public/protected constructors
singleton.no_known_relation ("C", "hasConstructor",
                                        OR("PublicConstructor", "ProtectedConstructor"));
singleton.retrieve("C");
var result =ontology.query(singleton);
```

Query 3-17 Final Singleton Pattern Query

In Query 3-17, the *no_known_relation* restriction states that the class C has *no* explicit modeled *hasConstructor* relationships with *PublicConstructor* or *ProtectedConstructor*, i.e. the class C has only private constructors. As DLs systems adopt an Open World Assumption (OWA) – any facts that are not modeled in a knowledge base cannot be proven. Therefore, simply using negated relational restriction as follows –

```
singleton.no _relation ("C", "hasConstructor", OR("PublicConstructor", "ProtectedConstructor"));
```

The query will not retrieve any result, since it is possible that class C has a public constructor or protected constructor, but they that are not specified in the knowledge base.

The *no_known_relation* restriction allows a localized Negation-As-Failure semantics to be applied to the query, so that the ontology reasoner can only answer the query based on the facts that are explicitly modeled in the knowledge base.

The above example illustrated how our software knowledge base can help programmers in recovering Design Patterns in the source code. We also illustrated how our approach can assist programmers in comprehending the intentions of software design, by either relaxing or tightening restrictions used in the queries. Users of our knowledge base are also able to define queries to identify patterns related to their specific comprehension tasks. In the next section, we show how our ontology-based approach supports programmers to reason about security concerns.

3.6.4. Reasoning about Security Concerns

Security vulnerabilities in a software system are typically caused by "carelessness or lack of awareness about *security concerns*" (Evans & Larochelle, 2002). With applications that become exposed to volatile environments with increased security risks (e.g., distributed environments, web centric applications, etc.), identifying security flaws in existing software systems becomes important activities in the software maintenance phase.

On top of various techniques to automatically identify software vulnerabilities at the source code level, manual code auditing is still considered a necessary. Manual auditing can indentify security flaws that would otherwise not be discovered through automatic

analysis (Evans & Larochelle, 2002). However, the cost associated with manual auditing is significantly higher compared to automatic analysis. This is mainly due to the fact that programmers have to understand what a security flaw might look like and then comprehend the code under review to be able to locate the flaw in the source code.

In this section, we discuss how our ontology- based comprehension approach can be applied to provide programmers or security experts with the ability to query and reason about various security concerns. Using our approach, programmers typically start with a hypothesis of what a security flaw looks like, and then apply static code reviewing to determine whether this security risk exists in the code. In this point of view, reasoning about a security concern is an iterative comprehension process based on examining and refining a hypothesis.

Object Accessibility

Typical Object Oriented Programming languages provide object modifiers such as *public*, *private*, and *protected* to restrict the accessibility of objects and methods. However, improper use of these access modifiers may cause security risks. For example, making class variables (fields) public may cause vulnerable data exposure and may cause undetermined behaviors if the public field has not been initialized.

Within our source ontology, we provide a set of atomic concepts and roles to facilitate querying and reasoning about such security concerns. In order to identify potential vulnerabilities caused by unexpected object accessibility, a user first creates a basic

scenario for the code auditing task by defining a public field as a field that has public modifier, i.e.

$PublicField \equiv Field \sqcap \exists\ hasModifier.PublicModifier$

This new concept can be defined through the *ontology* interface –

```
ontology.add_concept("PublicField", AND("Field", EXIST("hasModifier", "PublicModifer")));
```

Query 3-18 Define PublicField Concept

or alternatively, by using the scriptable query interface –

```
var SecurityConcern1 = new Query();
SecurityConcern1.declare("F", "MP");
SecurityConcern1.restrict("F", "Field");
SecurityConcern1.restrict("MP", "PublicModifier");
SecurityConcern1.restrict("F", "hasModifier", "MP");
SecurityConcern1.retrieve("F");
var result = ontology.query(SecurityConcern1);
```

Query 3-19 Public Fields

In this code auditing scenario, *allowing public and non-final fields in the code (indicating value of the field can be modified outside the class where it is defined) corresponds to a security risk*. Therefore, SecurityConcern1 can be further refined by adding the following constraints (Query 3-20) to retrieve all public fields without a final modifier –

```
SecurityConcern1.restrict("MF", "FinalModifier");
SecurityConcern1.no_relation("F", "hasModifier", "MF");
```

Query 3-20 Non-Final Public Fields

In order to extend the query for more specific tasks, such as: *Retrieve all public data of Java package "user.pkg1" that may potentially be accessed (read or write) by package "user.pkg2"*, users can further refine Query 3-20 by adding –

```
SecurityConcern1.restrict("F", "definedIn", "user.pkg1");
SecurityConcern1.restrict("M", "Method");
SecurityConcern1.restrict("M", "definedIn", "user.pkg2");
SecurityConcern1.restrict("M", "access", "F");
```

Query 3-21 Non-Final Public Fields Defined in user.pkg1

Reasoning services such as transitive closure and role classification are utilized in answering the above queries. For example, fields or methods in Java are defined in classes, and classes are defined in packages. The ontology reasoner will automatically determine the transitive relation *definedIn* between the concepts *Field/Method* and *Package*. In addition, read and write relationships between *Method* and *Variable* are modeled in our ontology by the *readField* and *writeField* roles, which are sub-roles of *access*.

Exception Handling

Exceptions correspond to events that can occur during the execution of a program that disrupt the normal flow of instructions. In Java, when an error occurs within a method, the method may throw an exception object that contains run-time information associated with the error. The caller method can then catch that exception and perform recovery from the error. While exception handling mechanisms have greatly simplified

error management, security concerns still may arise – an unhandled exception may cause programs to fail. Even worse, if such an exception occurs during file access, it may cause unexpected data exposure.

In Java, a method may arbitrarily throw a RuntimeException or any of its sub-classes, without necessarily being caught. For example, the *get* method of java.util.ArrayList in the Java JDK might throw an IndexOutOfBoundsException without forcing the caller method to catch that exception. Although runtime exceptions rarely occur, there are potential situations especially in multi-thread programs, when a situation like an object in one thread accesses the ArrayList object and at the same time its content may be modified by another thread. Therefore, this situation might lead to an unhandled exception. In the following example, we illustrate how our tool can be used to identify these types of problems caused by unhandled exceptions.

For example, the following query (Query 3-22) retrieves all methods that may throw RuntimeExceptions –

```
var SecurityConcern2 = new Query();
SecurityConcern2.restrict("M", "Method");
SecurityConcern2.restrict("E", "Exception");
SecurityConcern2.restrict("M", "throw", "E");
SecurityConcern2.restrict("E", "hasSuper", "java.lang.RuntimeException");
SecurityConcern2.retrieve("M");
var result = ontology.query(SecurityConcern2);
```

Query 3-22 Methods That Throw RuntimeException

The first two restrictions in Query 3-22 state that M and E are a Method and an Exception respectively. The third restriction states that method M throws an Exception E, and the last restriction expresses that E is a subclass of java.lang.RuntimeException. It has to be noted that the *hasSuper* role in our ontology represents the inheritance relationship between two classes. The transitivity of this role will be automatically handled by the ontology reasoner.

For a more detailed analysis, we extend the query further to *retrieve now only those methods that may invoke method M*, we can add –

SecurityConcern2.restrict("Caller", "Method");
SecurityConcern2.restrict("Caller", "invoke", "M");

and change the retrieval statement to –

SecurityConcern2.retrieve("Caller", "M");

Then, in order *to ensure all runtime exception thrown by M are handled (caught) by its Caller*, we could add one more restriction –

SecurityConcern2.no_relation("Caller", "catch", "E");

which restricts the result to be the complementary of Callers that catch exception E. In combination with other restrictions, the ontology reasoner will then retrieve each method that does not catch runtime exceptions a caller may throw. It also has to be

pointed out that by applying negated restrictions, potential performance issues may be introduced, thus their frequent use is not encouraged (Haarslev et al., 2004).

Security Enforcement

Many security flaws are preventable through security enforcement. Common vulnerabilities such as buffer overflows, accessing un-initialized variables, or leaving temporary files in the disk could be avoided by programmers with strong awareness of security concerns.

In order to deliver more secure software, many development teams have guidelines for coding practice to enforce security. Our knowledge base supports developers and security experts to enforce or validate whether these programming guidelines are followed. For example, to prevent access to un-initialized variables, a general guideline could be: *all fields must be initialized within class constructors.* The following query can retrieve all classes that did not follow this specific guideline –

```
var SecurityConcern3 = new Query();
SecurityConcern3.declare("F", "I", "C");
SecurityConcern3.restrict("F", "Field");
SecurityConcern3.restrict("I", "Constructor");
SecurityConcern3.restrict("C", "Class");
SecurityConcern3.restrict("F", "definedIn", "C");
SecurityConcern3.restrict("I", "definedIn", "C");
SecurityConcern3.no_relation("I", "writeField", "F");
SecurityConcern3.retrieve("C", "I");
```

Query 3-23 Constructors Didn't Initialize All Fields

In Java, all classes without a programmer defined constructor will have by default a no-argument constructor. These classes therefore can be initialized by any part of the program. A good security enforcement guideline is that *each class has to provide at least one constructor*. These classes without any constructors can be retrieved by the following query –

```
var SecurityConcern4 = new Query();
SecurityConcern4.declare("C");
SecurityConcern4.restrict("C", "Class");
SecurityConcern4.not_concept("C", KNOWN("hasConstructor"));
SecurityConcern4.retrieve("C");
```

Query 3-24 Classes Has No Constructors

The next example demonstrates how our knowledge base can enforce the following security practice: *all methods should take the responsibility to close the file(s) they have opened.* Such a guideline can be enforced by defining two new concepts such as *FileOpenMethod* and *FileCloseMethod*. These new concepts correspond to methods that are used to open or close files. Those methods have to be specified as instances of two concepts respectively. In a typical Java program, these methods include:

```
FileOpenMethod = {
java.io.File.createNewFile(),
java.io.File.createTempFile(),
java.io.FileInputStream.FileInputStream(),
…}
```

and

FileCloseMethod = {
java.io.FileInputStream.close(),
java.io.Writer.close(),
…}.

Based on these new two concepts, the following query can be applied to *detect potential methods that only invoke a file open method, but not a corresponding file close method*.

```
SecurityConcern5.restrict("M", "Method");
SecurityConcern5.restrict("O", "FileOpenMethod");
SecurityConcern5.restrict("C", "FileCloseMethod");
SecurityConcern5.restrict("M", "Invoke", "O");
SecurityConcern5.no_relation("M", "Invoke", "C");
SecurityConcern5.retrieve("M");
```

Query 3-25 Method Didn't Close Files it Opened

Query 3-25 also exposes some of the limitations of our ontology-based approach. At the current stage, the ontology language lacks the ability to represent temporal properties in the domain such as the sequence of actions. This might lead to situations where multiple occurrences of a relation are only captured once. In the SecurityConcern5 (file open/close) query, methods that invoke file open methods twice but call the corresponding close method only once are also returned. Furthermore, because our ontology is based on static source code analysis, the query cannot ensure the file

open/close methods are actually invoked or the proper sequence (first open then close) is followed during execution. However, from a code auditing and program comprehension perspective, our tool still provides valuable information to help auditors or security experts to identify flaws with regarding to specified security concerns.

Various static analysis approaches for detecting security vulnerabilities have been presented in literature, ranging from lexical (Bishop & Dilger, 1996), syntactical (Chess, 2002; Larochelle & Evans, 2001), and binary code (Hovemeyer & Pugh, 2004) flaw searching tools to more advanced model checking (Corbett et al., 2000) or theorem proving (Flanagan et al., 2002) approaches. Techniques for source code querying and searching are typically limited by their expressiveness and do not provide reasoning capabilities such as transitivity and classification. Other static analysis approaches include program verifiers or model checkers are dynamic techniques and normally very expensive and difficult to use (Evans & Larochelle, 2002).

3.6.5. Source-Document Links

Having both source code and documents represented in the form of an ontology allows us to link instances from source code and documentation using existing approaches from the ontology alignment domain (Noy & Stuckenschmidt, 2005). Ontology alignment techniques try to align ontological information from different sources on conceptual or/and instance levels. Since our documentation ontology and source code ontology share many concepts from the programming language domain, such as *Class* or *Method*, the problem of conceptual alignment has been minimized. This research

therefore focuses more on matching instances that have been discovered by both, our source analysis and text mining systems.

Our text mining system can additionally take advantage of the results from the source code analysis, by reusing them as input when detecting named entities. This allows us to directly connect instances from the source code and documents ontologies. For example, our source code analysis tool may identify c_1 and c_2 as classes, and this information can be used by the text mining system to identify named entities – c'_1 and c'_2 and their associated information in the documents (Figure 3-14). As a result, source code entities c_1 and c_2 are now linked to their occurrences in the documents (c'_1 and c'_2), as well as other information about the two entities mentioned in the document, such as design patterns, architectures, etc.

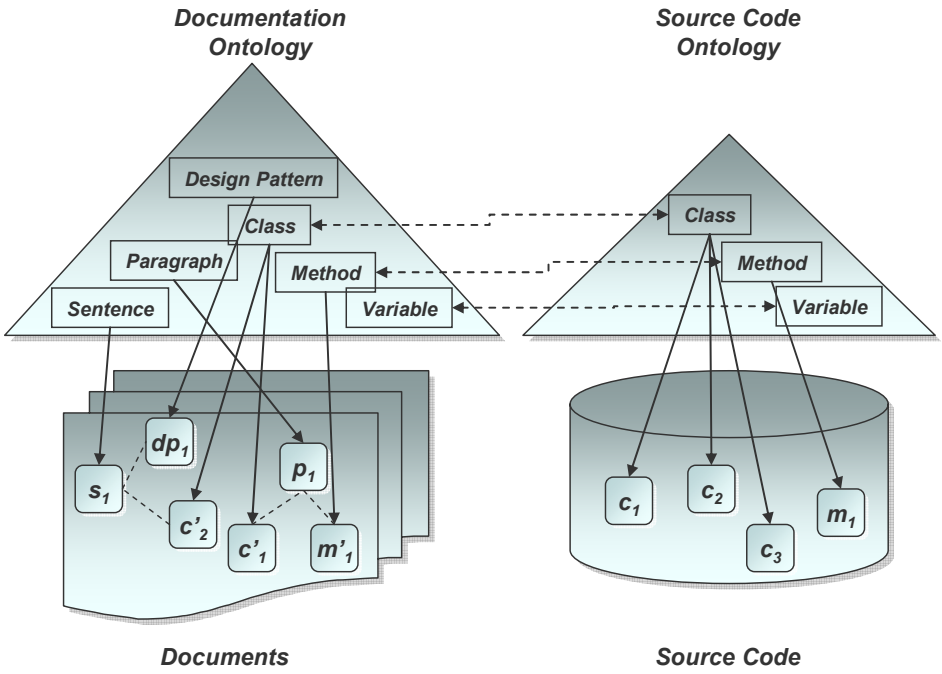

Figure 3-14 Linking Instances from Source Code and Documents

After source code and documentation ontology have been linked, users can then perform ontological queries on either documents or source code regarding properties of c_1 or c_2. For example – retrieve document passages that describe both c_1 and c_2, or retrieve design pattern descriptions referring to the class that contains the class currently analyzed. Note that the alignment process might also identify inconsistencies – the documentation might list a method for a different class, for example – which are detected through the alignment process and registered for further review by the user.

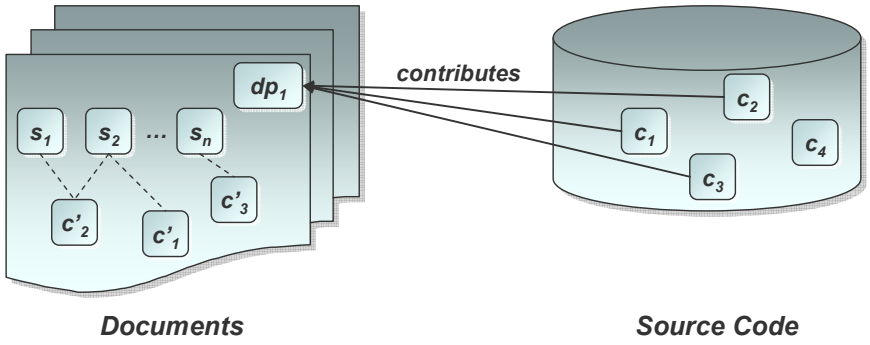

Figure 3-15 Retrieve Implicit Information from Documents

In addition, users can always manually define new concepts/instances and relationships in both ontologies to establish the links that cannot be detected by the automated alignment. For example, as Figure 3-15 shows, the text mining system may detect an instance of *DesignPattern* – dp_1 and users can then create the relationships between the pattern and classes that are contributed the pattern (e.g. c_1, c_2, and c_3) through our query interface. The newly created links then become an integrated part of the ontology, and can be used to, for example, retrieve all documents related to the pattern (i.e. s_1, s_2, …, s_n).

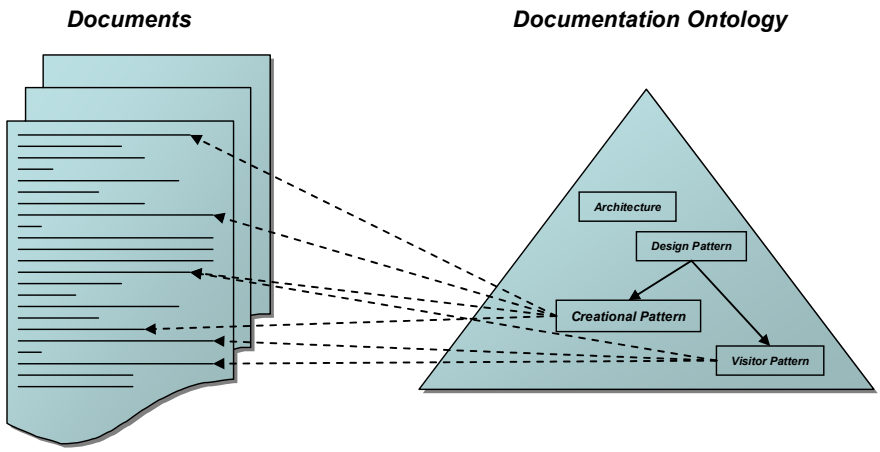

Figure 3-16 Classification of Documentation based on Ontology

Furthermore, documents can not only be linked to source code, but also to design-level concepts that relate to particular software maintenance tasks. For example, in contrast to the serialized view of software documents, i.e. sentence by sentence, or paragraph by paragraph, our formal ontological representation of software documentation also provides the ability to create hierarchical documentation views. Using the classification service of the ontology reasoner, one can classify document pieces that related to a specific concept or a set of concepts (Figure 3-16). For example, the Visitor Pattern (Gamma et al., 1995) documents can be considered as all text paragraphs that describe/contain information related to concept *Visitor* pattern. The following new concept *VisitorPatternDoc* can be used to retrieve paragraphs that related to the Visitor Pattern. Similarly, a new concept *HighlevelDoc* can be also defined to retrieve all documents that contains high level design concept *Architecture* or *DesignPattern*. The ontology reasoner can automatically classify documents according to these concept definitions.

$$VisitorPatternDoc \equiv Paragraph \sqcap \exists\ contains.Visitor$$

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

3.7. Contribution Summary

We have presented an ontology-based program comprehension model and its corresponding methodology to address the information searching and relating problems associated with a typical software comprehension process. The following contributions were presented in this research:

Cognitive Support

Inspired by mental model theory, which suggests the content of a mental model constitutes an ontology (Johnson-Laird, 1983), our ontological representations provide a hierarchical view of concepts and relationships existed in software artifacts. Such representations have a closer mapping to programmers' mental model, and thus can simplify the assimilation and accommodation process applied during program comprehension.

Constructive Learning

The ontological representations of software artifacts also allows programmers during source code and/or document reading, to actively constructing concepts by using logical constructors, such as conjunction, disjunction, or negation etc. Our contribution is supported by constructive learning theory (Piaget, 1971), which considers humans

actively learn new facts based on existing knowledge and by grouping its possibilities in different logical operations. Therefore, in contrast with existing approaches, which only focus on different representations for software systems, our approach also provide means for programmers to actively construct the mapping between their mental model and the software system.

Software Ontology

We present a formal ontological model for various software artifacts using DLs. DLs have several advantages over existing relational or ad-hoc models. Firstly, in terms of data schema, DLs are more expressive than relational models. In (Calvanese, Lenzerini, & Nardi, 1999), Calvanese et al proved that Entity-Relationship (ER) model can be translated to the DL language ***DLR*** (Calvanese, Giacomo, & Lenzerini, 1998) without loosing semantic information. Secondly, DLs allow for a precise specification of "IS-A" relationships between concepts, and thus are capable of capturing type hierarchies of programming languages. In addition, many state-of-the-art DLs reasoners (Haarslev & Möller, 2003) support the definition of transitive roles, which is very useful in reasoning transitive relations between source code elements. An example for such transitive relation is an "indirect-invocation" relationship between functions, which normally can not be expressed by relational models. The ontological model used in our approach also supports the use of role hierarchies, transitive roles, and concrete domain concept (e.g. line number of a source code element). The reasoning complexity of $\mathcal{ALCQHI}_{R^+}(\mathcal{D})$ logic – the core logic of Racer, is decidable.

Information Integration

Within our approach, we provide comprehensive concepts to capture the structural and semantic information in software document. The information captured from both source code and documents is integrated into one uniform ontological representation despite their languages (programming languages vs. natural languages). Information at different abstraction levels is associated according to the concept hierarchy defined in the ontologies. Moreover, common concepts that are shared between the source code and documentation ontology can help to reduce the conceptual gap between the artifacts they represent. Discovered concept instances from both source code and documents can be utilized to explore and establish links between these artifacts, and allow programmers to query properties of a software system across artifact boundaries (source code and documents).

Querying and Reasoning

Our approach allows programmers to query and navigate the software ontology using the nRQL query language provided by Racer. nRQL supports the formulation of conjunctive queries, in which query variables are used to be bound to instances that satisfy given restrictions. In order to support programmers in manipulating the query result, we introduced a scriptable query language based on JavaScript that allows programmers to benefit from both the declarative semantics of DLs, as well as the fine-grained control abilities of procedural languages.

In contrast to existing querying approaches that often provide a fixed vocabulary in their query languages, ontology-based approaches have the advantage of their extensible vocabularies. Complex concepts and relationships with regard to specific comprehension tasks can be defined on the fly. Such newly added concepts and relationships then become an integrated part of the ontology, and can be reused for further exploration/searching. An Ontology-based program comprehension approach also supports the decomposition of complex concepts and relationships into simpler ones. For example, in the case where higher level software elements, such as a component need to be analyzed, they can be specified using lower level concepts, e.g. package or class, by a "*contains"* relationship. These lower level concepts typically have direct correspondences with source code, and therefore ease the comprehension effort by automatically mapping complex concepts into their implementations.

Methodology

In addition to the ontology-based program comprehension model, we also provide a methodology that describes both the information required for understanding a software program, as well as the strategies to obtain this information. This methodology characterizes the program comprehension process as a set of activities corresponding to concept recognition (searching) and relationship discovery (relating), and thus can simplify many program comprehension tasks into ontology exploring and automated reasoning.

Applications

The benefits of adopting an ontology approach for program comprehension are directly related to the ease and support provided for a programmer to perform typical comprehension tasks. In this research, we provide application examples that demonstrate the applicability and flexibility of our approach. Furthermore, we also discuss the applicability, benefits and limitations of our approach for these application examples and compare them with existing approaches.

Chapter 4. SOUND – An ontology-based Program Comprehension Tool

In the previous chapter, we introduced our ontological software knowledge base, including its reasoning capability, its query interface, and several of its potential application domains. As part of this research, we have developed an ontology-based program comprehension environment – SOUND (Software Ontology for UNDerstanding), to support our ontology-based program comprehension approach. 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. In this chapter we discuss in detail implementation related issues of our environment. We first describe general requirements for the implementation of the knowledge base, followed by a description of the overall structure of our system. Next, we provide details on how we utilize information from source code analysis and text mining to automatically populate the software ontology.

4.1. Requirement

Based on the application examples discussed in Chapter 3, we can now derive implementation requirements for our ontological program comprehension approach.

- *Requirement #0*: Domain model. As a prerequisite, a sufficiently large part of the domain must be modeled in the form of an ontology, including the structural and semantic information of source code and documents to a level of detail that allows

relevant queries and reasoning on the properties of existing artifacts (e.g., for security analysis).

- *Requirement #1*: Ontology population. Program comprehension intrinsically needs to deal with a large number of artifacts from legacy systems. It is not feasible to manually create instance information for existing source code or documents. Thus, automatic ontology population methods must be provided for extracting semantic information from those artifacts.

- *Requirement #2*: Reasoning and querying. Software artifacts are often self-consistent. The structural and semantic integrity of an individual artifact can be maintained by specific tools, such as source code compiler. As a result, conceptual reasoning (i.e. TBox reasoning) has been minimized. However, for the ontological representation of software system one has to deal with a significant amount of identified instances and their relations are often implicit. Therefore, a highly efficient ontology reasoner that supports complex instance querying (i.e. ABox reasoning) is needed.

- *Requirement #3*: Integration with existing software development environments. The semantic ontological information must be accessible through a software maintainer's desktop. Knowledge obtained through querying and reasoning should be integrated with classical existing development tools and IDEs (e.g., *Eclipse*).

- *Requirement #4*: Applications. Finally, the adoption of formal ontologies in software maintenance is directly dependent on delivering added benefits to maintainers.

Simply providing a new technology without clearly showing how it can improve typical tasks, such as the ones described by our application examples in Section 3.6, will not lead to a high impact in the software maintenance community.

4.2. System Overview

In order to utilize the structural and semantic information in various software artifacts, we have developed an ontology-based program comprehension environment – SOUND, which can automatically extract concept instances and their relations from source code and documents (Figure 4-1).

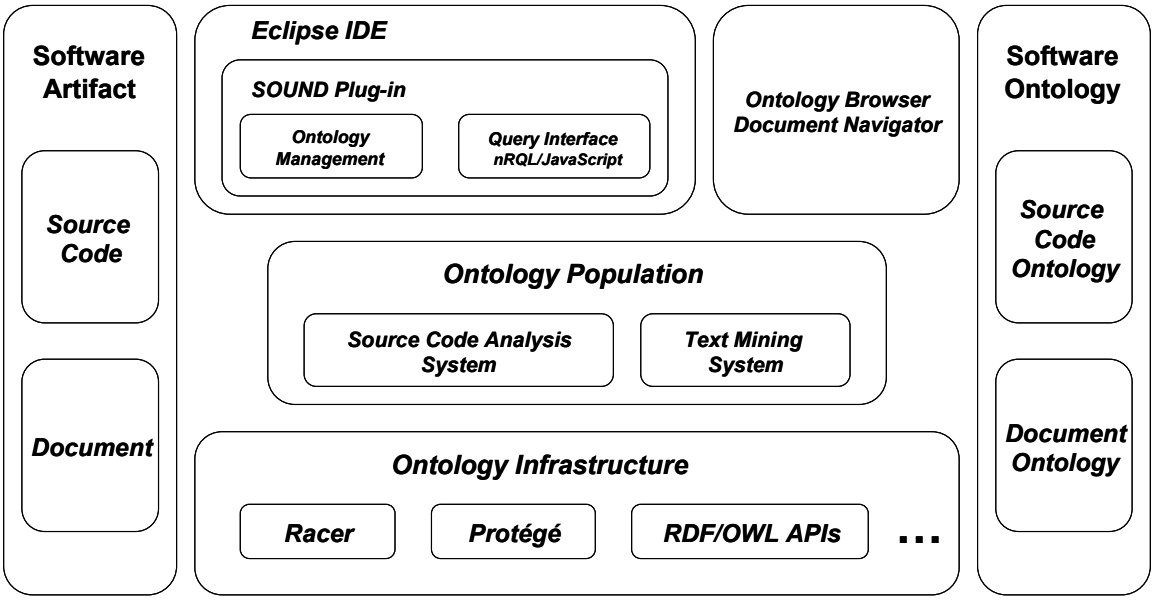

Figure 4-1 SOUND Environment Overview

As discussed earlier, an important part of our architecture is a *software ontology* that captures major concepts and relations in the software maintenance domain. This ontology consists of two sub-ontologies: a *source code* and *document* ontology, which

represent information extracted from source code and documents, respectively. The ontologies are modeled in OWL-DL (Smith, Welty, & McGuinness, 2004) and were created using the Protégé-OWL extension of Protégé – a free ontology editor.

Racer (Haarslev & Möller, 2003), an ontology inference engine, is adopted to provide reasoning services. The Racer system is a highly optimized DL system that supports reasoning about instances, which is particularly useful for the software maintance domain, where a large amount of instances needs to be handled efficiently.

Automatic ontology population is handled by two subsystems: The source code analysis, which is based on the JDT Java parser provided by Eclipse; and the document analysis, which is a text mining system based on the GATE (*General Architecture for Text Engineering)* framework (Cunningham, Maynard, Bontcheva, & Tablan, 2002).

The query interface of our system is a plug-in that provides OWL integration for Eclipse, a widely used software development platform. The expressive query language nRQL provided by Racer can be used to query and reason over the populated ontology. Additionally, we integrated a scripting language, which provides a set of built-in functions and classes using the JavaScript interpreter Rhino (http://www.mozilla.org/rhino/). This language simplifies querying the ontology for software engineers not familiar with DL-based formalisms.

4.3. 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 ontology technology in software maintenance. In this section, we describe in detail the automatic population of our ontologies from existing artifacts: source code (Section 4.3.1) and documents (Section 4.3.2).

4.3.1. Source Code Ontology

The source code ontology population subsystem is based on JDT – a Java language parser provided by Eclipse. JDT reads the source code and performs common tokenization and syntax analysis to produce an *Abstract Syntax Tree* (AST). Our population subsystem traverses the AST created by the JDT compiler to identify concept instances and their relations, which are then passed to an OWL generator for ontology population.

As an example (Figure 4-2), consider a single line of Java source code: *public int sort(){*, which declares a method called ***sort***.

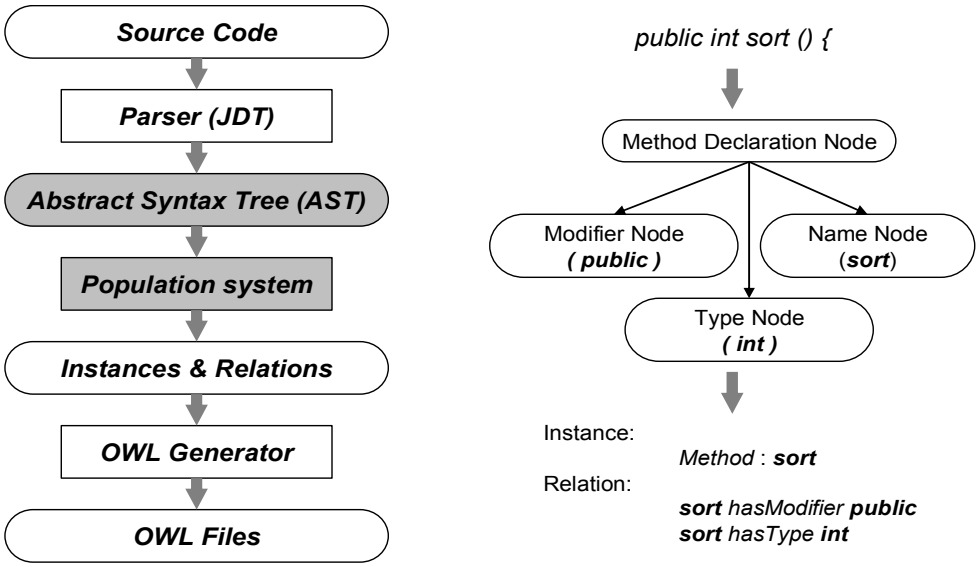

Figure 4-2 Populating Source Code Ontology

A simplified AST corresponding to this line of source code is shown in Figure 4-2. Our system traverses 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, the source code ontology contains 38 atomic concepts and 41 types of roles. We have performed several case studies on different open source systems to evaluate the size of the populated ontology. Table 4-1 summarizes the

results of our case studies, with the size of the software system being measured by lines of code (LOC) and the process time reflecting both AST traversal and ontology population.

Table 4-1 Source Code Ontology Size for Different Open Source Projects

	LOC	Proc. Time	Instances	Relations	Inst./LOC	Rel./LOC
java.util	24k	13.62s	10140	47009	0.42	1.96
InfoGlue[4]	40k	27.61s	15942	77417	0.40	1.94
Debrief[5]	140k	67.12s	52406	244403	0.37	1.75
uDig[6]	177k	82.26s	69627	284692	0.39	1.61

4.3.2. Documentation Ontology

We developed a custom text mining system to extract knowledge from software documents and populate the corresponding sub-ontology. The processing pipeline and its connection with the software documentation sub-ontology are shown in Figure 4-3. 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.

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

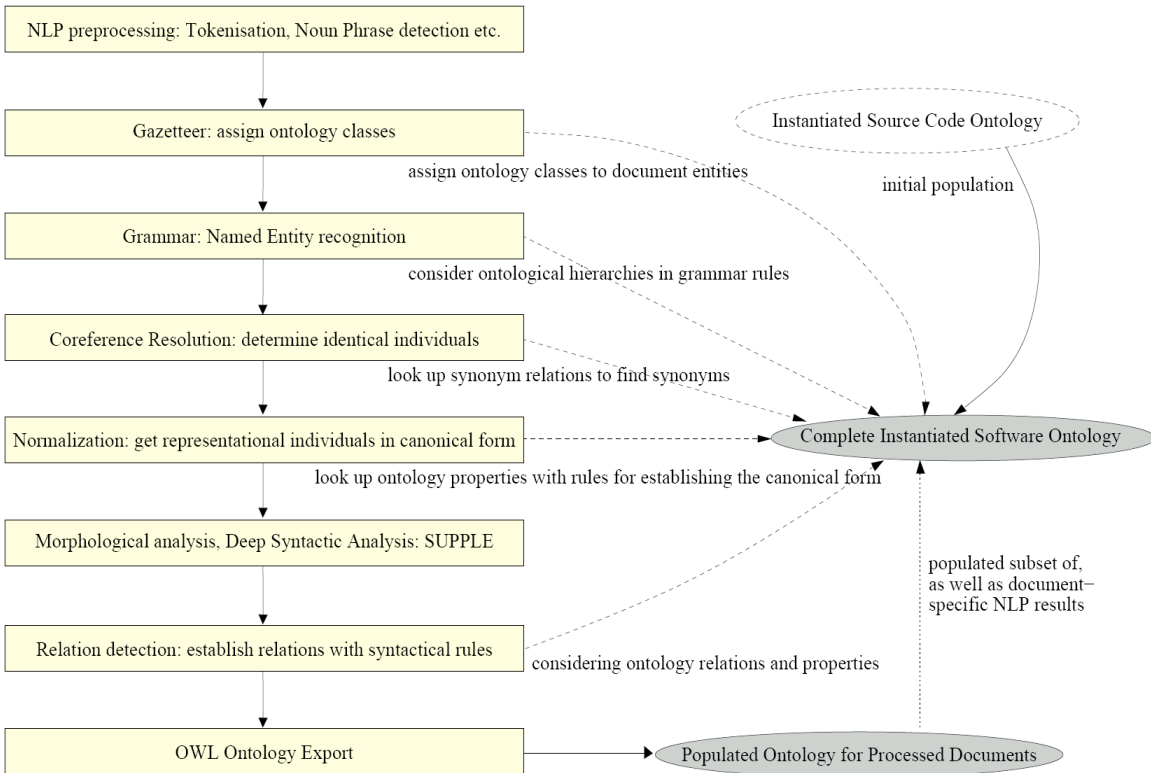

Figure 4-3 Workflow of the Ontology-Driven Text Mining Subsystem

The system first performs a number of standard preprocessing steps, such as tokenization, sentence splitting, part-of-speech tagging and noun phrase chunking (For more details, please refer to the GATE documentation: http://gate.ac.uk/documentation/). Then, named entities (NEs) modeled in the software ontology are detected in a two-step process: Firstly, an OntoGazetteer is used to annotate tokens with the corresponding class or classes in the software ontology (e.g., the word "architecture" would be labeled with the architecture class in the ontology). Complex named entities are then detected in the second step using a cascade of finite-state transducers implementing custom grammar rules written in the JAPE language, which is part of GATE. These rules refer back to the annotations generated by the

OntoGazetteer, and also evaluate the ontology. For example, in a comparison like class=="Keyword", the ontological hierarchy will be utilized to ensure that a JavaKeyword will also result in a match, since a Java keyword is-a keyword in the ontology. This significantly reduces the overhead for grammar development and testing (Witte, Kappler, & Baker, 2007).

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 (Witte, Li, Zhang, & Rilling, 2007).

The next step is the detection of relations between the identified entities in order to compute predicate-argument structures, like implements (class, interface). In this step two different and largely complementary approaches are combined: 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.

Chapter 5. Case Study

5.1. Architectural Analysis

A variety of software 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 the application of our SOUND environment to support the architecture analysis. The software under study, InfoGlue (http://www.infoglue.org/), 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.

For the case study we applied the following four major steps, which are corresponds to our comprehension methodology introduced in Section 3.5: (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.

5.1.1. Identifying Architectural Styles

The first step of an architecture analysis is typically to identify potential architectural styles (Shaw & David, 1996) 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.

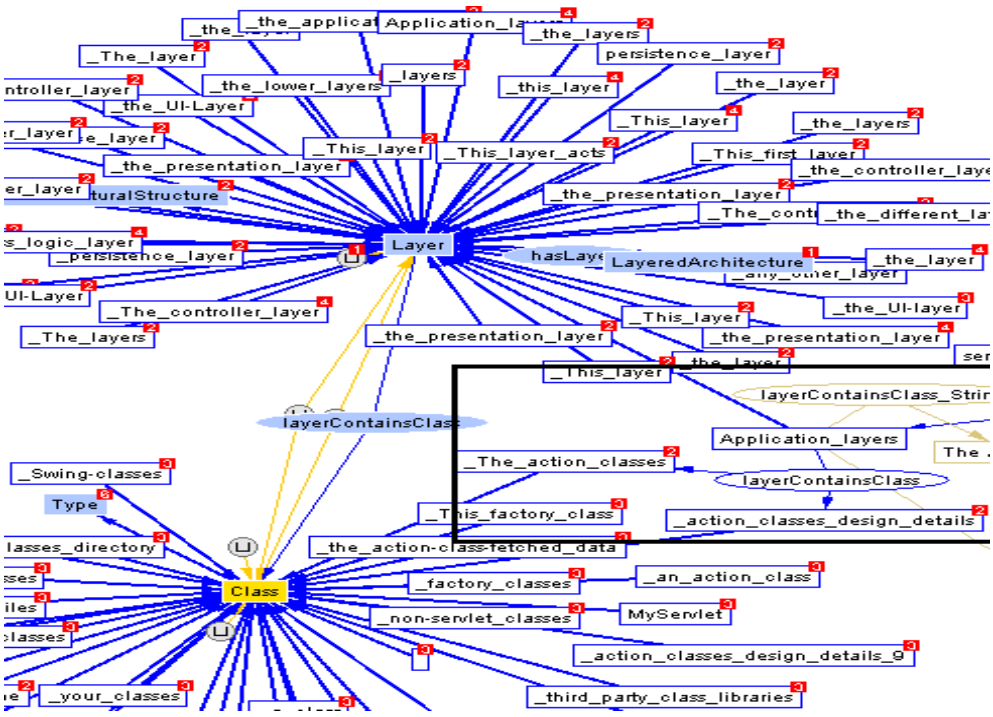

Figure 5-1 Documentation Ontology Populated Through Text Mining

In order to identify the architectural style of the InfoGlue system, we first use our Text Mining system to analyze the architecture document and populate the documentation ontology. The documentation ontology contains various concepts concerning software architecture, such as MVC, Layered Architecture, or Pipeline-and-Filters (Shaw & David, 1996). By browsing the populated ontology, we observe that a large number of instances of concept *Layer* (Figure 5-1) are discovered. This information provides us with significant clues that the InfoGlue system might be implemented using a typical Layered Architecture (Shaw & David, 1996).

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 using 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 5-2.

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

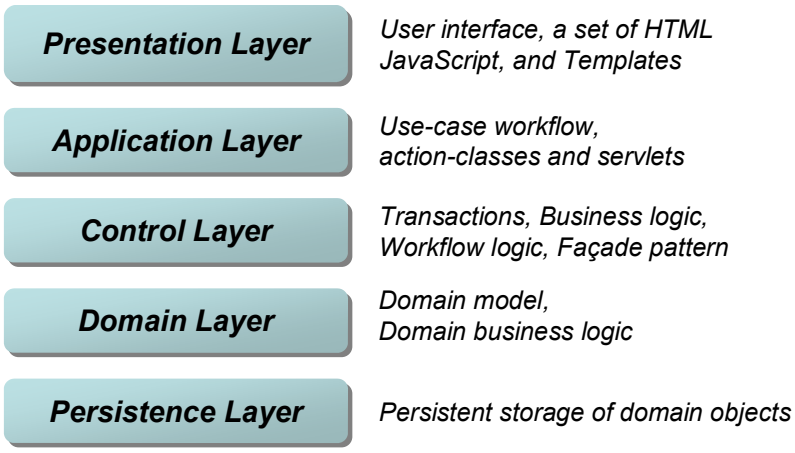

Figure 5-2 Layers in the InfoGlue System

5.1.2. 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");
```

Query 5-1 Definition of Layers

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 5-3. This information provides important references for our further analysis of the documents and source code.

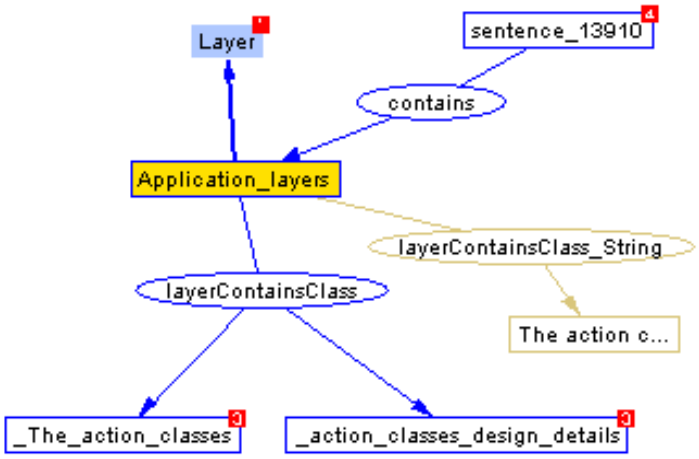

Figure 5-3 Semantic Information Discovered by Text Mining

Next we find, by searching source code, 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");
}
```

Query 5-2 Action Classes are Defined in Application Layer

It has to be noted that in case when newly acquired concepts are needed for other comprehension tasks, users of the SOUND environment can define and create 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, *SuperClassOfAction*, and *ActionClass*:

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

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

Query 5-3 Definition of Action Classes

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

$$ActionClass \equiv Class \sqcap \exists hasSuperType.SuperClassOfAction$$

Our reasoner can automatically infer instances of *ActionClass* through this definition. As previously discussed, this newly created concept 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.

5.1.3. Reasoning about Component Properties

In steps 3 and 4 of the architecture analysis, we 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.

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 (Shaw & David, 1996). 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.");
        }
}
```

Query 5-4 Retrieve Method Calls between Layers

The script first retrieves all layer instances in the ontology, and then iteratively queries method call relationship between layers. A similar query is performed to retrieve the number of methods being called. Figure 5-4 summarizes the results of these two queries, indicating the number of method calls and the number of methods being called.

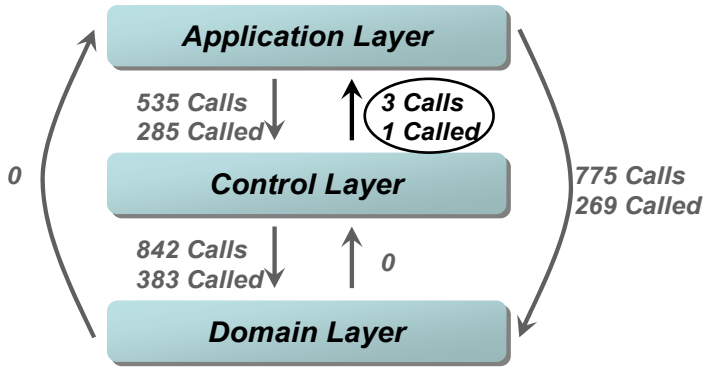

Figure 5-4 Method Calls between Layers

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 3 method calls from the control layer to the application layer can be considered as either implementation defects or as a 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.

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 5-5 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.

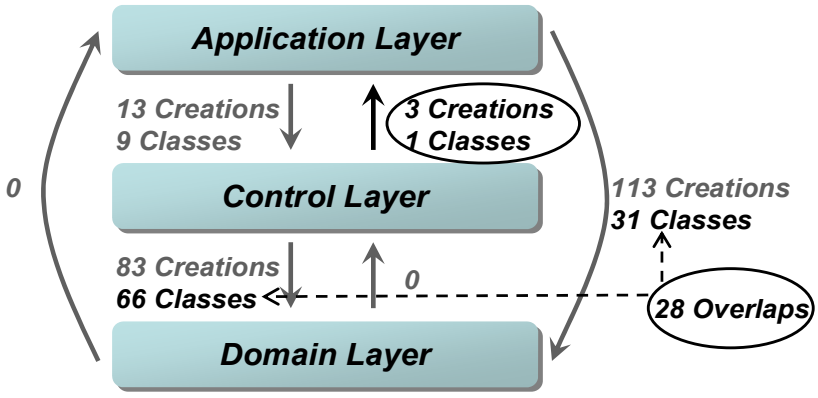

Figure 5-5 Object Creations between Layers

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());
```

Query 5-5 Domain Object created by both Application and Control Layers

The result of this script indicates 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.

Design Patterns

The hypotheses concerning properties of different components are not only motivated based on a maintainer's expertise, but also based on information obtained from the documentation ontology. For example, our documentation ontology also discovered instances of "façade pattern" (Gamma et al., 1995) in the architecture document. We also found that these instances are mostly associated with the control layer. Therefore, we hypothesized that the control layer may implement a façade pattern.

```
var query = new Query();
query.declare("C", "M1", "M2");
query.restrict("C", "Class");
query.restrict("M1", "Method");
query.restrict("M2", "Method");
query.restrict("M1", "definedIn", "control_layer");
query.restrict("M1", "definedIn", "C");
query.restrict("M2", "definedIn", "application_layer");
query.restrict("M2", "call", "M1");
query.retrieve("C");
var result = ontology.query(query);
out.println(result);
```

Query 5-6 Detect Façade Pattern in Control Layer

In order to validate our hypothesis, Query 5-6 was used to retrieve classes in the control layer that are accessed by the application layer. The retrieved classes are therefore public access points of the control layer – corresponding to the façade classes. During

the maintenance of the control layer, changes to these classes have to be performed with special care, since improper modifications may cause interface incompatibility.

The result of this query shows that 146 out of 295 classes in the control layer correspond to façade classes. From these results, we suspected that the document provides us with incorrect information about the use of the façade pattern. This is due to the fact that the implementations of these façade classes provide the control layer with too much exposure. This is contrary to the context where a façade pattern is typically applied (limiting the exposure of the subsystem). Our further analysis shows that these "façade classes" actually implement the typical Bridge (Gamma et al., 1995) patterns. Therefore, we conclude that either the document is inconsistent with the source code, or the designers of the InfoGlue system have their own specific understanding of the purpose and context of a façade pattern.

5.1.4. Summary and Evaluation

In this section, we presented a case study that focused on the use of our SOUND environment to comprehend a software system at the architectural level. The case study clearly demonstrates how the Text Mining subsystem 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 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*.

We have also performed initial experiments to evaluate the execution time of the scripts introduced in this section. The InfoGlue system contains about 64K lines of source code and 770 classes distributed in 49 Java packages. The experiments are performed on a Pentium M 1.6Ghz computer with 512MB memory. Our SOUND system discovered 15917 individuals and 83691 relations in the InfoGlue. Table 5-1 summarizes the execution results.

Table 5-1 Execution Time of Queries

Script/Query	Execution
Control Communication between Layers	23 seconds
Data Communication between Layers	11 seconds
Overlaps of Object Creations	<1 second
Façade Pattern	<1 second

5.2. Traceability Links

As part of this case study, we have extended our SOUND environment by several ontology alignment rules to link the documentation ontology and source code ontology. An initial evaluation has been performed on a large open source Geographic

Information System (GIS) – uDig[7]. The uDig system is a set of Eclipse Plug-ins that provides geographic information management integration for Eclipse platform. The uDig documents used in the study consist of a set of JavaDoc files and a requirement analysis document.[8]

5.2.1. Experiment

Links between the uDig implementation and its documentation are recovered by first performing source code analysis to populate the source code ontology. The resulted ontology contains instances of *Class*, *Method*, *Field*, etc, and their relationships such as inheritance, invocation, etc. Our text mining system 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, architecture styles etc. The ontology alignment rules are then applied to link both the documentation ontology and the source code ontology. Parts of our initial results are shown in Figure 5-6, and the contents of related sentences are:

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

[7] http://udig.refractions.net/confluence/display/UDIG/Home
[8] http://udig.refractions.net/docs/

Sentence_712: *"Use the visitor pattern to traverse the AST"*

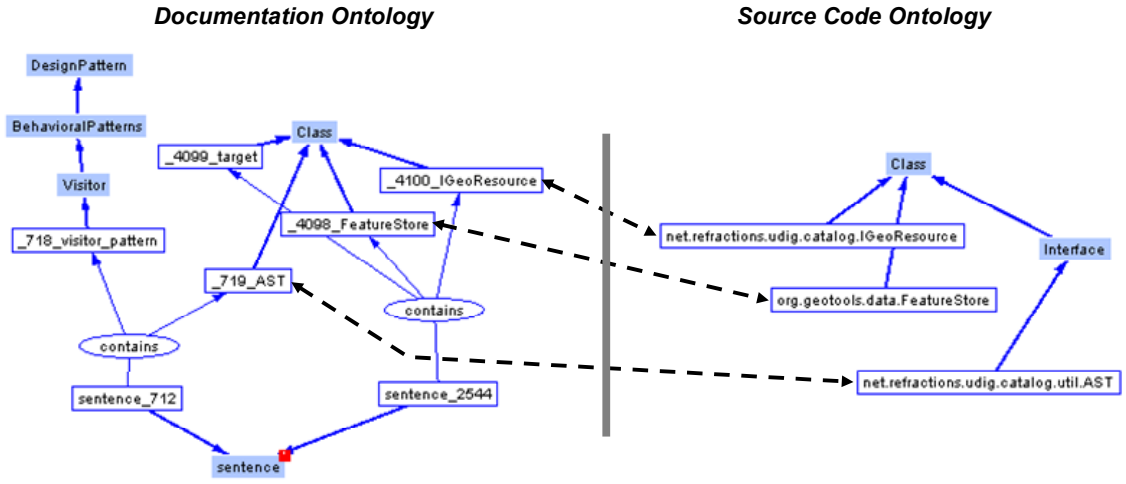

Figure 5-6 Linked Source Code and Documentation Ontology

Figure 5-6 shows that in the uDig documents our text mining system discovers that a sentence (*sentence_2544*) contains both class instance *_4098_FeatureStore* and *_4100_IGeoResource*, which can be linked to the corresponding 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 instance (*_719_AST*) and a design pattern instance (*_718_visitor_pattern*) are also identified. Instance *_719_AST* can then be linked to the *net.refractions.udig.catalog.util.AST* interface in the source code ontology in a similar manner.

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. For example, during the comprehension of the class *FeatureStore*, a reverse engineer may want to study the classes that are related to *FeatureStore*. Within the source code ontology, a query similar to the Query 5-7 can be performed to retrieve all classes that contain methods that have called 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
```

Query 5-7 Query on Source Code Ontology

Unfortunately the class *IGeoResource*, which has a documented relation with *FeatureStore* (Figure 5-6), will not be returned by such a query, because *IGeoResource* has no explicit invocation relations with *FeatureStore* in the uDig implementation. In addition the maintainer can perform source code queries across the boundaries between source code and documentation. Such type of queries is enabled due to the already established links between the source code and documentation ontology. For

example, the following query (Query 5-8) retrieves all classes that occur in the same sentences as class *FeatureStore*. At 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
```

Query 5-8 Query on Documentation Ontology

The linked source code and documentation ontologies also provide the capability to combine semantic information from both software implementation and documentation. For example, the text mining system has detected that class *AST* is potentially a part of a Visitor pattern (Figure 5-6). In order to retrieves all documented information related to the detected pattern, the following query (Query 5-9) 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
```

Query 5-9 Query across the Source Code and Documentation Ontology

This query utilizes both, the programming language semantics, such as the inheritance relationship between query variable *C* and the class *AST*, and the structural information of documentation, such as the containing relationship between *P* and *C*. The result of this query contains all text paragraphs that describe the sub classes of *AST*, i.e. the Vistor pattern. It has to be noted that the role *contains* is a transitive relation to describe the document structure. The ontology reasoner automatically resolves the transitivity from *Paragraph* to *Sentence*, and from *Sentence* to *Class*.

5.2.2. Evaluation

The case study provided an initial evaluation of recovering traceability links between source code and documentation on a large open source software system. We have demonstrated the use of automated reasoning to retrieve documented information with regard to a specific reverse engineering task and infer implicit relationships in the linked ontologies. So far, we evaluated our system on two collections of texts: a set of 5 documents (7743 words) from the Java documentation for the Collections[9] framework and a set of 7 documents (3656 words) from the documentation of the uDig system. The

[9] http://java.sun.com/j2se/1.4.2/docs/guide/collections/

document sets were chosen because of the availability of their corresponding source code.

Both sets were manually annotated for named entities, including their ontology classes and normalized form, as well as relations between the entities. In what follows, we present results from the named entity (NE) recognition, entity normalization, and relation detection tasks.

- *Named Entity Recognition*

We computed the standard precision, recall, and F-measure results for NE detection. A named entity was only counted as correct if it matched both the textual description and ontology class.

Table 5-2 shows the results for two experiments: first running only the text mining system over the corpora (left side) and second, performing the same evaluation after running the code analysis, using the populated source code ontology as an additional resource for NE detection as mention earlier.

Table 5-2 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%

**P=Precision, R=Recall, F=F-measure, A=Accuracy*

As shown in Table 5-2, the text mining system achieves a very high precision (90%) in the NE detection task, with a recall of 62%. With the imported source code instances, 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 utilizing 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 source code analysis, which are currently under development. However, the current results already underline the feasibility of our approach of integrating code analysis and NLP.

- *Entity Normalization Evaluation*

We also evaluated the performance of our lexical normalization rules for entity normalization, since correctly normalized names are 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 5-2 shows the results for both the system running in text mining mode alone and with additional source code analysis. As shown in the table, the normalization component performs rather well.

- *Relation Detection Evaluation*

Relation detection is typically the hardest subtask within a text mining system. Like for entity detection, we performed two different experiments, with and without source code analysis results. Additionally, we evaluated the influence of the semantic relation filtering step using our ontology as described above. The results are summarized in Table 5-3.

Table 5-3 Relation Detection Performance*

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

** P=Precision, R=Recall, F=F-measure*

Table 5-3 shows that the current combination of rules with the SUPPLE parser does not achieve a high performance. However, the increase in precision (ΔP) when applying the

filtering step using our ontology is significant: a 54% improvement in precision compared to the prior results without semantic filtering.

The overall low precision and recall values are mainly due to the unchanged SUPPLE parser rules, which have not yet been adapted to the software domain. Also, the conservative PP-attachment strategy of SUPPLE misses main predicate-argument structures. We currently experiment with different parsers (RASP and MiniPar) and are also adapting the SUPPLE grammar rules in order to improve the detection of predicate-argument structures.

Chapter 6. Conclusions and Future Work

As discussed throughout the thesis, program comprehension is a knowledge intensive activity that requires a large amount of effort to synthesize information obtained from different sources. These resources include among others, source code, comments, specification, maintenance logs, and etc. The main goal of the presented research is to investigate the applicability of ontologies in program comprehension, and thereby, to promote the use of semantic and knowledge representation techniques in software maintenance.

The presented ontology-based program comprehension model provides consistent ontological representations for various software artifacts, such as source code and software documents. In particular, we have developed a formal source code ontology to represent the complex structure of software implementation, and a documentation ontology to capture concepts and relations discovered from software documents. We also address the challenge of ontology population, by providing automatic ontology population for both ontologies. The source code ontology is populated through source code parsing and for the automatic population of the documentation ontology text mining techniques have been applied.

An ontology-based comprehension methodology has been introduced and defined to describe both, the information required for comprehending a software program, as well

as the strategies to obtain the information. This methodology characterizes the program comprehension process as a set of activities that correspond to concept recognition (searching) and relationship discovery (relating). The methodology provides maintainers with guidance on how program comprehension tasks can be mapped to ontological explorations.

We discussed several applications and case studies to illustrate how these typical comprehension/maintenance tasks can be accomplished within our ontological comprehension methodology. These applications and case studies clearly demonstrated the applicability and flexibility of our approach. In particular, the ontology-based program comprehension approach can reduce the comprehension effort by automatically identifying concept instances and their relationships found in different software artifacts. The resulting source code ontology and documentation ontology provide means of mapping semantic information discovered from documentation to implementation and vice versa. Furthermore, we have adopted a top-down comprehension strategy in one of the case studies, to demonstrate that our approach supports the formulations of hypotheses concerning properties of a software system, and that ontological reasoning services can be applied to validate these hypotheses.

We were also able to show that our approach can simplify the *assimilation* process applied during program comprehension, since a human's learning process also contains an *accommodation* part that may lead to a reorganization of the existing knowledge (Piaget, 1971). As part of our ongoing work, we intend to introduce non-monotonic

reasoning and fuzzy Description Logics (Baader, 2003) to further facilitate this accommodation process.

The source code and documentation ontologies are currently 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 the understanding of large code and document bases, we improved the precision of the text mining subsystem by supporting both broader and semantically richer queries. As part of future work we plan to integrate support for the explicit representation of uncertain and incomplete information through a fuzzy set theory-based approach similar to (Kölsch & Witte, 2003) to enrich the information represented in our current DL-based knowledge repository.

The presented research promotes the use of both formal ontology and automated reasoning in program comprehension. The ontological representations for various software artifacts also provide closer mappings to a programmer's knowledge, and therefore ease the construction of appropriate mental models of software programs. Through the exploration of a ontological linking strategy that involves source code, including inline comments (like JavaDoc), over implementation, design, and specification documents to domain-specific knowledge, we envision to be able to offer in the future a truly holistic process for an automated support of program comprehension.

Bibliography

Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (1979). awk – A Pattern Scanning and Processing Language. *Software Practice and Experience,* 9(9), pp. 267-280.

Anderson, P., & Zarins, M. (2005). The CodeSurfer software understanding platform, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 147-148.

Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., & Welty, C. (2006). Supporting Online Problem-solving Communities With the Semantic Web, *The 15th International Conference on World Wide Web (WWW'06)*, pp.575-584

Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2000). Information retrieval models for recovering traceability links between code and documentation, *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 40-49.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2000). Tracing object-oriented code into functional requirements, *Proceedings of the 8th IEEE International Workshop on Program Comprehension (IWPC'00)*, pp. 79-86.

Antoniol, G., Casazza, G., di Penta, M., & Merlo, E. (2001). A method to re-organize legacy systems via concept analysis, *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pp. 281-290.

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software, *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pp. 153-160.

Arnold, R., & Bohner, S. (2003). *Software Change Impact Analysis*: Wiley-IEEE Computer Society.

Baader, F. (2003). *The description logic handbook - theory, implementation, and applications*. Cambridge University Press.

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

Beck, K., & Johnson, R. (1994). Patterns Generate Architectures, *Proceedings of the European Conference on Object-Oriented programming (ECOOP'94)*, pp. 139-149.

Berardi, D., Calvanese, D., & Giacomo, G. D. (2003). *Reasoning on UML Class Diagrams using Description Logic Based Systems*. Università di Roma.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. (1993). The concept assignment problem in program understanding, *Proceedings of the 15th International Conference on Software Engineering,* pp. 482-498.

Bishop, M., & Dilger, M. (1996). Checking for Race Conditions in File Accesses. *Computing Systems,* 9(2), pp. 131-152.

Bohner, S. A., & Arnold, R. S. (1996). S*oftware Change Impact Analysis*. IEEE Computer Society Press.

Brachman, R. J., Devanbu, P., Selfridge, P. G., Belanger, D., & Chen, Y. (1990). Toward a Software Information System. *AT&T Technical Journal,* 69(2), pp. 22-41.

Brachman, R. J., & Levesque, H. J. (1985). *Readings in Knowledge Representation*. Los Altos, CA, M. Kaufmann Publishers.

Brin, S., & Page, L. (1998). The Anatomy of a Large-scale Hypertextual Web Search Engine, *The seventh international conference on World Wide Web*.

Brooks, F. P. (1987). No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer,* 20(4), pp. 10-20.

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies,* 18(6), pp. 542-554.

Bruce, K. B., & Wegner, P. (1990). An Algebraic Model of Subtype and Inheritance. In F. Bancilhon & P. Buneman (Eds.), *Advances in Database Programming Languages, Frontier Series*, pp. 75--96. New York, NY, ACM Press.

Bull, R. I., Trevors, A., Malton, A. J., & Godfrey, M. W. (2002). Semantic grep: Regular Expressions + Relational Abstraction, *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02),* pp. 267-276.

Calvanese, D., Giacomo, G. D., & Lenzerini, M. (1998). On the Decidability of Query Containment Under Constraints, *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Seattle, Washington, United States, ACM Press.

Calvanese, D., Lenzerini, M., & Nardi, D. (1999). Unifying Class-Based Representation Formalisms. *Journal of Artificial Intelligence Research,* 11, pp. 199-240.

Chen, Y. F., Nishimoto, M. Y., & Ramamoorthy, C. V. (1990). The C information Abstraction System. *IEEE Transactions on Software Engineering,* 16(3), pp. 325-334.

Chess, B. (2002). Improving Computer Security using Extended Static Checking, *IEEE Symposium on Security and Privacy.* IEEE CS Press.

Chikofsky, E. J., & Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software,* 7(1), pp. 13-17.

Clarke, C. L. A., & Cormack, G. V. (1996). *Context grep*, Department of Computer Science, University of Waterloo.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., & Robby, a. H. Z. (2000). Bandera: Extracting Finite-state Models from Java Source Code, The *22nd International Conference on Software Engineering*. Limerick, Ireland.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental Representations of Expert Procedural and Object-oriented Programmers in a Software Maintenance Task. *International Journal of Human-Computer Study,* 50(1), pp. 61-83.

Cox, A., & Collard, M. L. (2005). Textual Views of Source Code to Support Comprehension, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 109-112.

Cunningham, H., Maynard, D., Bontcheva, K., & Tablan, V. (2002). GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications, *40th Meeting of the Association for Computational Linguistics (ACL'02)*. Philadelphia, PA.

Decker, B., Rech, J., Ras, E., Klein, B., & Hoecht, C. (2005). Selforganized Reuse of Software Engineering Knowledge Supported by Semantic Wikis, *Workshop on Semantic Web Enabled Software Engineering*.

Devanbu, P. T., Brachman, R. J., Selfridge, P. G., & Ballard, B. W. (1990). LaSSIE - a Knowledge-based Software Information System, *Proceedings of the 12th international conference on Software engineering*. Nice, France

Devanbu, P. T., & Jones, M. A. (1994). The Use of Description Logics in KBSE systems, *The 16th International Conference on Software Engineering*. Sorrento, Italy.

Ebert, J., Kullbach, B., & Winter, A. (1999). GraX - An Interchange Format for Reengineering Tools, *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pp. 89-98.

Evans, D., & Larochelle, D. (2002). Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software,* 19(1).

Exton, C. (2002). Constructivism and Program Comprehension Strategies, *Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pp. 281-284.

Feijs, L., Krikhaar, R., & Ommering, R. v. (1998). A Relational Approach to Support Software Architecture Analysis. *Software - Practice and Experience,* 28(4), pp. 371-400.

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

Fischer, M., Oberleitner, J., Gall, H., & Gschwind, T. (2005). System Evolution Tracking Through Execution Trace Analysis, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 237-246.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. (2002). Extended Static Checking for Java, *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns : elements of reusable object-oriented software*. Reading, MA, Addison Wesley Professional.

Grass, J. E. (1992). Object-Oriented Design Archaeology with CIA++. *Computing Systems: The Journal of the USENIX Association,* 5(1), pp. 5-67.

Gutwin, C., Penner, R., & K. Schneider. (2004). Group Awareness in Distributed Software Development, *ACM Conference on Computer Supported Cooperative Work*.

Haarslev, V., & Möller, R. (2003). Racer: A Core Inference Engine for the Semantic Web, *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON'03)*, pp. 27-36.

Haarslev, V., Möller, R., & Wessel, M. (2005). Description Logic Inference Technology: Lessions Learned in the Trenches, *The 2005 International Workshop on Description Logics*. Edinburgh, Scotland, UK.

Haarslev, V., Moller, R., & Wessel, M. (2004). Querying the Semantic Web with Racer + nRQL, *KI-2004 International Workshop on Applications of Description Logics (ADL'04)*.

Hall, P. H. (1992). *DATALOG. Environmental fate data bases*: Boca Raton, Syracuse Research Corp. Lewis Publishers.

Happel, H. J., Korthaus, A., Seedorf, S., & Tomczyk, P. (2006). KOntoR: An Ontology-enabled Approach to Software Reuse, *The 18th Int. Conf. on Software Engineering and Knowledge Engineering*.

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

Hayes, P. J. (1979). The logic of frames. In D. Metzing (Ed.), *Frame Conceptions and Text Understanding*, pp. 46-61. Walter de Gruyter and Co.

Hecht, M. S. (1977). *Flow Analysis of Computer Programs*. North Holland, New York, Elsevier Science Ltd.

Hendrickson, S. A., Dashofy, E. M., & Taylor, R. N. (2005). An (Architecture-centric) Approach for Tracing, Organizing, and Understanding Events in Event-based Software Architectures, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 227-236.

Hohmann, L. (1996). *Journey of the Software Professional: The Sociology of Software Development*.

Holt, R. C. (1996). Binary Relational Algebra Applied to Software Architecture, CSRI Technical Report 345, University of Toronto.

Horrocks, I., Kutz, O., & Sattler, U. (2006). The Even More Irresistible SROIQ, *The 10th International Conference of Knowledge Representation and Reasoning.* Lake District, UK.

Horrocks, I., Sattler, U., & Tobies, S. (2000). Reasoning with Individuals for the Description Logic SHIQ, *17th International Conference on Automated Deduction (CADE 2000)*.

Hovemeyer, D., & Pugh, W. (2004). Finding Bugs is Easy, *Companion of the 19th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* ACM Press.

Ferrante Jeanne, Ottenstein J. Karl, & Warren D. Joe (1987). The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS),* 9(3), pp. 319-349.

Jin, D., & Cordy, J. R. (2003). A Service Sharing Approach to Integrating Program Comprehension Tools, *European Software Engineering Conference*.

Johnson-Laird, P. N. (1983). *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Cambridge, MA, Harvard University Press.

Joy, B., Steele, G., Gosling, J., & Bracha, G. (2000). *Java Language Specification (2nd Edition)*, Prentice Hall PTR.

Juergen Rilling, Yonggang Zhang, Wenjun Meng, René Witte, Volker Haarslev, & Philippe Charland (2006), A Unified Ontology-Based Process Model for Software Maintenance and Comprehension, *Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'06)*. Springer, LNCS.

Juergen Rilling, René Witte, & Yonggang Zhang (2007), Automatic Traceability Recovery: An Ontological Approach, *International Symposium on Grand Challenges in Traceability (GCT'07)*. Center of Excellence in Traceability. Lexington, Kentucky, USA.

Kölsch, U., & Witte, R. (2003). Fuzzy Extensions for Reverse Engineering Repository Models, *10th Working Conference on Reverse Engineering (WCRE)*. Victoria, Canada.

Keller, R. K., Schauer, R., Robitaille, S., & Lague, B. (2001). *The SPOOL Approach to Pattern Based Recovery of Design Components*, Springer-Verlag.

Keller, R. K., Schauer, R., Robitaille, S., & Page, P. (1999). Pattern-based Reverse Engineering of Design Components, *21st International Conference on Software Engineering*.

Khan, L., & McLeod, D. (2000). Audio Structuring and Personalized Retrieval Using Ontologies, *IEEE advances in digital libraries*, Library of congress.

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

Klint, P. (2003). How Understanding and Restructuring Differ from Compiling - a Rewriting Perspective, *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 2-11.

Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Revevant Informaiton during Software Maintenance Tasks. *IEEE Transactions of Software Engineering,* 32(12), pp. 971-987.

Koenemann, J., & Robertson, S. (1991). Expert Problem Solving Strategies for Program Comprehension, *Human Factors in Computing Systems*, ACM Press.

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

Kullbach, B. (2000). *Command Line GReQL: CLG*: University of Koblenz-Landau, German.

Kullbach, B., & Winter, A. (1999). Querying as an Enabling Technology in Software Reengineering, *The 3rd EuroMicro Conference on Software Maintenance and Reengineering,* pp. 42.

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

Lange, C., Sneed, H. M., & Winter, A. (2001). Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools, *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pp. 209-218.

Larochelle, D., & Evans, D. (2001). Statically Detecting Likely Buffer Overflow Vulnerabilities, *10th Usenix Security Symp.(USENIX'01)*, Usenix Assoc.

Lethbridge, T. C., & Anquetil, N. (1997). *Architecture of Source Code Exploration Tools: A Software Engineering Case Study*. University of Ottawa.

Letovsky, S. (1986). *Cognitive Processes in Program Comprehension*. Ablex Publishing Corp.

Linton, M. A. (1984). Implementing Relational Views of Programs, *ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Enviroment*.

Möller, R. (1996). A Functional Layer for Description Logics: Knowledge Representation Meets Object-Oriented Programming, *OOPSLA´96*. San Jose, California.

Magloire, A. (2000). *Grep: Searching for a Pattern*. Iuniverse Inc.

Marcus, A., & Maletic, J. I. (2002). Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing, *25th International Conference on Software Engineering*.

Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., & Sergeyev, A. (2005). Static Techniques for Concept Location in Object-oriented Code, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 33-42.

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

Mayank, V., Kositsyna, N., & Austin, M. (2004). *Requirements Engineering and the Semantic Web Part II - Representation, Management and Validation of Requirements and System-Level Architectures* (No. TR 2004-14). University of Maryland.

McClure, C. (1992). *The Three R's of Software Automation: Re-Engineering Repository Reusability*. NJ, Prentice Hall.

McGuinness, D. L., & Harmelen, F. v. (2004). *OWL Web Ontology Language Overview*, http://www.w3.org/TR/owl-features/

Wenjun Meng, Juergen Rilling, Yonggang Zhang, René Witte, Sudhir Mudur, Philippe Charland (2006), A Context-Driven Software Comprehension Process Model, *Second International IEEE Workshop on Software Evolvability (SE'06),* pp. 50-57

Wenjun Meng, Juergen Rilling, Yonggang Zhang, René Witte, & Philippe Charland (2006). An Ontological Software Comprehension Process Model, *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM'06)*. Genoa, Italy.

Minsky, M. L. (1974). *A Framework for Representing Knowledge*: Cambridge, Massachusetts Institute of Technology A.I. Laboratory.

Muller, H., & Klashinsky, K. (1998). *Rigi - A System for Programming-in-the-large.* Paper presented at the 10thInternational Conference on Software Engineering (ICSE'98).

Murphy, G. (1996). *Lightweight Structural Summarization as an Aid to Software Evolution.* University of Washington.

Murphy, G., Notkin, D., & Sullivan, K. (1995). Software Reflexion Models: Bridging the Gap between Source and High-Level Models, *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

Nelson, M. L. (1996). *A Survey of Reverse Engineering and Program Comprehensio*, NASA.

Nonenmann, U., & Eddy, J. K. (1992). KITSS – A Functional Software Testing System Using a Hybrid Domain Model, *The 8th IEEE Conference on Artifical Intelligence Applications*.

Novak, J. D. (1998). *Learning, Creating, and Using Knowledge: Concept Maps(tm) As Facilitative Tools in Schools and Corporations*. LEA, Inc.

Noy, N. F., & Stuckenschmidt, H. (2005). *Ontology Alignment: An Annotated Bibliography – Semantic Interoperability and Integration*. Schloss Dagstuhl, Germany.

Pagan, F. G. (1991). *Partial Computation and the Construction of Language Processors*, Prentice Hall.

Patel-Schneider, P. F., McGuiness, D. L., Brachman, R. J., Resnick, L. A., & Borgida, A. (1991). The CLASSIC Knowledge Representation System: Guiding Principles and Implementation Rational. *SIGART Bull,* 2(3), pp. 108–113.

Paul, S., & Prakash, A. (1994a). A Framework for Source Code Search Using Program Patterns. *IEEE Transactions of Software Engineering,* 20(6).

Paul, S., & Prakash, A. (1994b). Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*.

Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology,* 19, pp. 295-341.

Piaget, J. (1971). *Science of Education and the Psychology of the Child*. Viking Press.

Premkumar, D., Ronald, B., Peter, G. S., & Bruce (1991). LaSSIE: A Knowledge-based Software Information System. *Communications of the ACM,* 34(5), pp. 34-49.

Quillian, M. R. (1967). Word Concepts: A Theory and Simulation of Some Basic Capabilities. *Behavioral Science,* 12, pp. 410–430.

Racer. (2005). *RacerPro User's Guide (Ver. 1.9)*, Racer System GmbH& Co. KG.

Rajlich, V., & Wilde, N. (2002). The Role of Concepts in Program Comprehension, *Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pp. 271-278.

Reiss, S. P. (2005). Efficient Monitoring and Display of Thread State in Java, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 247-256.

Revelle, M., Broadbent, T., & Coppit, D. (2005). Understanding Concerns in Software: Insights Gained from Two Case Studies, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 23-32.

Rich, C., & Waters, R. (1990). *The Programmer's Apprentice*. Baltimore, Maryland, Addison-Wesley.

Rugaber, S. (1995). Program Comprehension. *Encyclopedia of Computer Science and Technology*.

Sartipi, K., & Kontogiannis, K. (2001). A Graph Pattern Matching Approach to Software Architecture Recovery, *proceedings of IEEE International Conference on Software Maintenance,* pp. 408-419.

Schmidt-Schauß, M., & Smolka, G. (1991). Attributive Concept Descriptions with Complements. *Artificial Intelligence,* 48(1), pp. 1-26.

Schneiderman, B. (1986). An Empirical Studies of Programmers, *2nd Workshop on Empirical Studies of Programmers*. Ablex Publishers.

Selfridge, P. G. (1991). Knowledge Representation Support for a Software Information System, *The Seventh Conference on Artificial Intelligence Applications*.

Selfridge, P. G. (1992). Knowledge-Based Software Engineering. *Journal of IEEE Intelligent Systems,* 7(6), pp. 11-12.

Selfridge, P. G., & Heineman, G. (1994). Graphical Support for Code-level Software Understanding, *the 9th Conference on Knowledge-Based Software Engineering (KBSE'94).* IEEE Computer Society.

Shaw, M., & David, G. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Shneiderman, B. (1980). *Software Psychology: Human Factors in Computer and Informaiton Systems*. Winthrop Publishers. Inc.

Shneiderman, B., & Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences,* 8(3), pp. 219-238.

Sim, S. E. (1998). *Supporting Multiple Program Comprehension Strategies During Software Maintenance.* University of Toronto, Toronto.

Singer, J., & Lethbridge, T. C. (1997). *What is so great about grep?* University of Ottawa.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2006). Pellet: A Practical OWL-DL Reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2), pp. 51-53.

Smith, M. K., Welty, C., & McGuinness, D. L. (2004). *OWL Web Ontology Language Guide*.

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE transactions on Software Engineering,* 10(5), pp. 595-609.

Soloway, E., Pinto, J., Letovsky, S., Littman, D., & Lampert, R. (1988). Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM,* 31(11), pp. 1259-1267.

Sowa, J. F. (1999). *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Course Technology.

Stelzner, M., & Williams, M. D. (1988). *The Evolution of Interface Requirements for Expert Systems*. Norwood, NJ. Ablex Publishing.

Storey, M. A. (2005). Theories, Methods and Tools in Program Comprehension: Past, Present and Future, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 181-191.

Sun, D., & Wong, K. (2005). On Evaluating the Layout of UML Class Diagrams for Program comprehension, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 317-326.

Synytskyy, N., Holt, R. C., & Davis, I. (2005). Browsing Software Architectures with LSEdit, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 176-178.

Tilley, S. R., Paul, S., & Smith, D. B. (1996). Towards a Framework for Program Understanding, *4th International Workshop on Program Comprehension*.

Tjortjis, C., Sinos, L., & Layzell, P. (2003). Facilitating Program Comprehension by Mining Association Rules from Source Code, *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 125-132.

Tsarkov, D., & Horrocks, I. (2006). FaCT++ Description Logic Reasoner: System Description, *International Joint Conference on Automated Reasoning (IJCAR 2006)*.

Von Mayrhauser, A., & Vans, A. M. (1994). Comprehension Processes During Large Scale Maintenance, *16th International Conference on Software Engineering*.

Von Mayrhauser, A., & Vans, A. M. (1995). Program Comprehension During Software Maintenance and Evolution. *IEEE Computer,* 28(8), pp. 44-55.

Walenstein, A. (2002). *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework.* PhD thesis. Simon Fraser University.

Walkinshaw, N., Roper, M., & Wood, M. (2005). Understanding Object-oriented Source Code from the Behavioral Perspective, *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pp. 215-224.

Weiser, M. (1981). Program Slicing, *5th International Conference on Software Engineering*. San Diego, California.

Weiser, M. (1984). Program Slicing. *IEEE Transactions on Software Engineering,* 10(4), pp. 352-357.

Welty, C. (1995a). *An Integrated Representation for Software Development and Discovery.* Vassar College, Poughkeepsie, NY.

Welty, C. (1995b). Towards and Epistemology for Software Representations, *10th Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press.

Welty, C. (1996). An HTML interface for CLASSIC, *The 1996 Description Logic Workshop (DL'96),* AAAI Press/The MIT Press.

Welty, C. (1997). Augmenting Abstract Syntax Trees for Program Understanding, *The 1997 International Conference on Automated Software Engineering*, IEEE Computer Society Press.

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.

Wills, L. M. (1994). Using Attributed Flow Graph Parsing to Recognize Programs, *Workshop on Graph Grammars and Their Application to Computer Science*.

Witte, R., Kappler, T., & Baker, C. J. O. (2006). Ontology Design for Biomedical Text Mining. In *Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences,* Springer Verlag.

Witte, R., Kappler, T., & Baker, C. J. O. (2007). Ontology Design for Biomedical Text Mining. In *Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences*, Springer Verlag.

Witte, R., Li, Q., Zhang, Y., & Rilling, J. (2007). Ontological Text Mining of Software Documents, *12th International Conference on Applications of Natural Language to Information Systems (NLDB 2007)*. Paris, France.

Witte R., Li, Q., Zhang, Y., & Rilling, J. Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach. *To appear, IET Software Journal, Special Issue on Natural Language in Software Development*.

René Witte, Yonggang Zhang, and Juergen Rilling (2007). Empowering Software Maintainers with Semantic Web Technologies, *4th European Semantic Web Conference (ESWC'07)*.Innsbruck, Austria.

Wongthongtham, P., Elizabeth Chang, Tharam S. Dillon, & Ian Sommerville (2005). Software Engineering Ontologies and Their Implementation, IASTED Conference on Software Engineering, pp. 208-213

Wouters, B., Deridder, D., & Paesschen, E. V. (2000). The Use of Ontologies as a Backbone for Use Case Management, *European Conference on Object-Oriented Programming - Workshop on Objects and Classifications, a Natural Convergence*.

Wu, S., & Manber, U. (1992). Agrep - A Fast Approximately Pattern-Matching Tool, *USENIX Winter 1992 Technical Conference*. San Francisco, U.S.A.

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

Yonggang Zhang, Juergen Rilling, & Volker Haarslev (2006). An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns, *30th Annual International Computer Software and Applications Conference* (COMPSAC'06), pp. 333-342

Yonggang Zhang, Rene Witte, Juergen Rilling, & Volker Haarslev (2006). Ontology-based Program Comprehension Tool Supporting Website Architectural Evolution, *Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*, pp. 41-49

Yonggang Zhang, René Witte, Jürgen Rilling, & Volker Haarslev (2006). An Ontology-based Approach for the Recovery of Traceability Links, *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM'06)*. Genoa, Italy.