EXTRACTING SEMANTICS FROM LEGACY SOURCES USING REVERSE ENGINEERING OF JAVA CODE WITH THE HELP OF VISITOR PATTERNS

By

OGUZHAN TOPSAKAL

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

To My Mother H. Nedret Topsakal
And
To My Father Sabahatdin Topsakal

ACKNOWLEDGMENTS

I would like to give my sincere thankfulness to my thesis advisor, Dr. Joachim Hammer. His kindness, encouragement and thoughtfulness meant so much during this thesis study. I also would like to thank Dr. Douglas Dankel and Dr. Beverly Sanders for serving on my committee. I also would like to thank all of my colleagues in the Scalable Extraction of Enterprise Knowledge (SEEK) project, especially Dr. Mark Schmalz, Lucian, Ming Xi, Nikhil and Sangeetha.

I am grateful to my parents, H. Nedret Topsakal and Sabahatdin Topsakal; to my brother, Metehan Topsakal; and to my sister-in-law, Sibel Topsakal. They were always there when I needed them; and they support me in whatever I do.

I also would like to thank all of my friends, my special treasures, for affecting my life positively. There are some friends who have showed their care the most in the past two years. H. Levent Akyil and Saskia Kameron are the ones who encouraged me the most in pursuing advanced study in the U.S.; and supported me in every aspect during my stay in U.S. My dear friend, Evren Dilbaz was always with me although he is thousands of miles away. Members of *Hayallerimiz* team, Bahri Okuroglu and Oguzhan Poyrazoglu, have always amazed me with their friendship. A. Koksal Hocaoglu, Nihat Kasap and Aylin Yilmaz are the new treasures I have gained in Gainesville.

I would like to acknowledge the National Science Foundation for supporting this research under grant numbers CMS-0075407 and CMS-0122193.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES x

ABSTRACT xii

CHAPTER

1 INTRODUCTION 1

1.1 Motivation 2
1.2 Solution Approaches 3
1.3 Challenges 4
1.4 Contributions 5
1.5 Organization of Thesis 6

2 RELATED RESEARCH 8

2.1 Code Reverse Engineering 9
2.1.1 What is a Decompiler? 9
2.1.2 History of Decompilers 10
2.1.3 Why Successful Decompiling is Possible in Java 10
2.1.4 Purpose of Decompilation 11
2.1.5 Examples of Decompilers 11
2.1.5.1 Mocha 11
2.1.5.2 Jad 12
2.2 Parser Generation 12
2.2.1 Compiler-Compiler Tools 13
2.2.2 JavaCC and its Features 13
2.2.3 The Grammar Specification File of JavaCC 16
2.2.4 Java Tree Builder (JTB) 16
2.3 Visitor Design Pattern 19
2.3.1 Design Patterns 19
2.3.2 Features of Visitor Design Pattern 19
2.3.3 Analogy for Visitor Pattern 20
2.3.4 Double Dispatching in Visitor Patterns 21

2.4 State-of-the-art in Extracting Knowledge from Application Code....22
2.4.1 Lexical and Syntactic Analysis....22
2.4.2 Control Flow Analysis....22
2.4.3 Program Slicing....23
2.4.4 Pattern Matching....23
2.5 Business Rules Extraction....23
2.5.1 Business Rules....24
2.5.2 Categories of Business Rules....24
2.5.3 Where Are Business Rules Implemented in Information Systems....25
2.5.4 Extracting Business Rules....26
2.5.5 Importance of Business Rule Extraction from Program Code....26

3 SEMANTIC ANALYSIS ALGORITHM....27

3.1 Introduction....27
3.2 Objective of Semantic Analysis....30
3.3 Components of the Semantic Analyzer....31
3.3.1 Decompiler....31
3.3.2 Abstract Syntax Tree Generator....32
3.3.3 Information Extractor....33
3.4 Extensibility of the Semantic Analyzer....34
3.5 Extracting Semantics from Multiple Files....35
3.6 Finding the Class Source Code Files....36
3.6 Slicing Variables....37
3.7 Heuristics Used....38

4 IMPLEMENTATION OF THE JAVA SEMANTIC ANALYZER....44

4.1 Package Structure....45
4.2 Using the Java Tree Builder To Augment the Grammar Specification File....48
4.3 Generate Parser by JavaCC....49
4.4 Execution Steps of Java Semantic Analyzer....50
4.4.1 Making Preparations to Analyze....51
4.4.2 Analyze Source Code....52
4.5 Functionality of Visitor Classes....54
4.5.1 Output Abstract Syntax Tree to XML File....54
4.5.2 Get Variable and Method Declaration Information....55
4.5.3 Get Slicing Information....56
4.5.4 Get Business Rules and Comments....57
4.5.6 Get Method Calls....60
4.5.7 Get Package-Import-Extend Info....61
4.5.8 Get SQL Statements....61
4.5.9 Parse SQL Strings....61

5 EVALUATION OF SEMANTIC ANALYZER 63

5.1 Can Features of Semantic Analyzer Extract Semantics Correctly? 63
5.1.1 Targeting a Specific Column from the Query 63
5.1.2 Extracting the Column Name from Mathematical Expression 65
5.1.3 Geting Query Expression from the Variable Value 66
5.1.4 Concatenating Strings to Get the Query Expression 67
5.1.5 Geting the Meaning of Slicing Variable from the Output Text 68
5.1.6 Passing Semantics from one Method to Another 69
5.1.7 Extracting the Meaning from the Title Output 69
5.2 Correctness Evaluation 71
5.3 Performance Evaluation 75

6 CONCLUSION AND OPPORTUNITIES FOR FUTURE RESEARCH 78

6.1 Contributions 79
6.2 Future Work 80

APPENDIX

A GRAMMAR SPECIFICATION FILE EXAMPLE FOR JAVACC 83

B COMPILER BASIC TERMS 85

B.1 Production Rules 85
B.2 LL and LR Grammars 85
B.3 Table Driven Top-Down Parsing 85
B.4 Lexical Analyzer 86
B.5 The Parser 86

C DECOMPILED FILE EXAMPLE 87

D VISITOR CLASS EXAMPLE 92

E JAVA GRAMMAR 96

F ORACLE SQL GRAMMAR 101

G CONSTRUCTION SCHEDULING PROGRAM 104

LIST OF REFERENCES 115

BIOGRAPHICAL SKETCH 117

LIST OF TABLES

Table page

2-1 Automatically generated tree node interface and classes 17

4-1 JavaCC generated parser files. 49

4-2 Information stored for variable/method declaration in SymbolTableElement. 57

4-3 Information stored in BusinessRule class. 58

4-4 Information stored in ResultsetSQL structure 62

5-1 Examples of ResultSet get method parameters. 64

5-2 Example of a SQL query that has mathematical expression in its column list. 65

5-3 Example of using variable in the *executeQuery* method. 66

5-4 Example of Concatenating String to Get the Query Expression. 67

5-4 Making relations between output statements and columns. 68

5-5 An example of passing parameter values. 70

5-6 Extracting the meaning from the title. 70

5-7 Extracted Meanings of the Columns from the Construction Scheduling Program Source Code. 72

5-8 Extracting the meaning from title of the outputs. 73

5-9 Code fragment inside the main method of ScheduleMenu.java 74

5-10 Code fragment from ScheduleList.java shows how variable semantic is transferred from one method to another. 74

5-11 Time spent (in seconds) in sub steps of previous version of SA. 76

5-12 Time spent (in milliseconds) in current version of SA. 77

A-1 Example of .jj file. 84

C.1 Original version of ScheduleMenu.java code. 87
C.2 De-compiled version of ScheduleMenu.java code. 89
D.1 Class Definitions of Example Nodes of Tree 93
D.2 Visitor Interface 93
D.3 Visitor Class Example 94
G.1 Code of ScheduleMenu.java. 104
G.2 Code of ScheduleList.java 106
G.3 Code of ManipulateActivity.java. 107

LIST OF FIGURES

Figure page

3-1 High-level view of the SEEK Architecture. 27

3-2 Semantic Analyzer Components. 31

3-3 Checking the success of decompiling process. 32

3-4 Two main steps of Information Extraction. 33

3-5 Traversing between class files. 36

3-6 File Finder 37

4-1 Architectural overview of the SA prototype and related components. 44

4-2 Java Semantic Analyzer Packages Structure 1. 46

4-3 Java Semantic Analyzer Packages Structure 2. 47

4-4 Input and outputs of the JTB tool. 49

4-5 Initialization of the Java Semantic Analyzer. 51

4-6 Eight steps that make up *prepareForProcess* method. 51

4-7 Seven steps that make up *analyzeMethods* method. 53

4-8 Link List Structure of Traversed Method Tree of SA. 55

4-9 Example of a business rule that inherits another business rule. 58

4-10 The content of the stacks at point when "*" in Figure 4-9 is reached. 59

4-11 The content of the stacks at point when '#' in Figure 4-9 is reached 60

4-12 List of Method Calls Structure. 60

4-13 Example of a ResultsetSQL class. 62

5-1 Example of a *ResultsetSQL* class and *ColumnElement* and *TableElement* lists. 65

5-2 ResultSetSQL class representation of the query in Table5.2. 66

5-3 Illustration of String *SymbolTableElement*s and their values. 67

D-1 Example of production rules of Java grammar. 92

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

EXTRACTING SEMANTICS FROM LEGACY SOURCES
USING REVERSE ENGINEERING OF JAVA CODE
WITH THE HELP OF VISITOR PATTERNS

By

Oguzhan Topsakal

August 2003

Chair: Joachim Hammer
Major Department: Computer and Information Science and Engineering

As the need increase for enterprises to participate in large business networks (e.g., supply chains), the need to optimize these networks to ensure profitability also increases. However, because of the heterogeneities of the underlying legacy information systems, existing integration techniques fall short of enabling the automated sharing of data among participating enterprises. Current integration techniques require manual effort and significant programmatic set-up. This necessitates the development of more automated solutions to enable scalable extraction of knowledge resident in legacy systems of a business network, to support efficient sharing. Given that an application is a rich source for semantic information including business rules, we developed algorithms and methodologies to extract semantic knowledge from the legacy application code.

Our novel methodology integrates and improves existing techniques including program slicing, program dependence graphs, pattern matching, and design patterns. Our

techniques advance the state-of-the art in many ways, most importantly to bring a generic approach that can be applied to any language code.

Semantic knowledge extracted from the legacy application code contains information about the application-specific meaning of entities and their attributes as well as business rules and constraints. Once extracted, this semantic knowledge is important to the schema matching and wrapper generation processes.

This thesis presents an overview of our approach. Evidence to demonstrate the extraction power and features of this approach is presented using a prototype developed in the context of the Scalable Extraction of Enterprise Knowledge (SEEK) project at the Database Research and Development Center at the University of Florida.

CHAPTER 1
INTRODUCTION

In today's business markets, many factors increase the competition such as customization of products, demand on rapid delivery, and decrease in profit. To survive in this competitive environment, efficient collaborative production in a business network is required. An enterprise or business network is the collaboration of several firms to achieve a common goal. The success of a business network is highly dependent upon coordination among participants. Decision and integration support tools are needed to improve the ability of a participant in a business network to coordinate, plan, and respond to dynamically changing conditions. Because of the heterogeneity of the legacy information systems of the participants, data and knowledge integration among systems in a business network require a great deal of programmatic setup with limited code reusability. Therefore, there is a need to develop a toolkit that can semi-automatically discover enterprise knowledge from sources; and use this knowledge to configure itself and act as software between the legacy sources. The Scalable Extraction of Enterprise Knowledge[1] (SEEK) project underway in the Database Research and Development Center at the University of Florida is directed at developing methodologies to overcome some of the problems of assembling knowledge in numerous legacy information systems [1,2].

[1] This project is supported by National Science Foundation under grant numbers CMS-0075407 and CMS-0122193.

1.1 Motivation

A legacy source is defined as a complex stand-alone system with poor or nonexistent documentation for the data and application code or other components of the information system. A typical enterprise network has contractors and subcontractors that use such legacy sources to manage their data. The data present in these legacy sources is an important input to making decisions at the project level. Legacy systems of participants of enterprise network usually have a high degree of physical and semantic differentiations for the reasons stated below. These differences cause many problems for accessing and retrieving data from the underlying legacy source to provide input to decision support tools. The problems of providing information for decision support tools [1] can be stated as follows.

- There is a difference between various internal data storage, retrieval, and representation methods. Some firms might use professional database management systems while some others might use simple flat files to store and represent their data. There are many interfaces including SQL or other proprietary languages that a firm may use to manipulate its data. Because of such high degrees of physical heterogeneity, the effort to retrieve even similar information from every firm in the supply chain requires extensive work on the data stored in each system.

- There is semantic heterogeneity among firms. Even though the supply chain consists of firms working in the closely related domains, there is a difference in the internal terminology. This kind of heterogeneity presents at various levels in the legacy information source including the application code that may encode business rules.

- Another challenge in accessing the firm's data efficiently and accurately is safeguarding the data against loss and unauthorized usage. Participating firms in the supply chain may want to restrict the sharing of strategic knowledge including sensitive data or business rules. Therefore, it is important to develop third-party tools that assure the privacy of the concerned firm and still extract useful knowledge.

- The frequent need of human intervention in the existing solutions is another major problem for efficient cooperation. This limits automation of the process which makes it costly and inefficient.

Thus, it is necessary to build extensible data access and extraction technology that automates the knowledge extraction process.

1.2 Solution Approaches

The role of the SEEK system is to act as an intermediary between the legacy data and the decision support tool. In general, SEEK [1-3] operates as a three-step process:

Step 1: SEEK generates a detailed description of the legacy source including application-specific meanings of the entities, relationships, and business rules. We collectively refer to this information as enterprise knowledge. Extracting enterprise knowledge is a build-time process which is achieved by the data reverse engineering (DRE) component of SEEK. The Schema Extractor (SE), which is a sub-component of DRE, extracts the underlying database conceptual schema while another sub-component of DRE, the Semantic Analyzer (SA), extracts application specific meanings of the entities, attributes, and the business rules of the enterprise.

Step 2: The semantically enhanced legacy source schema must be mapped onto the domain model (DM) used by the applications that want to access the legacy source. This is done using a schema mapping process that produces the mapping rules between the legacy source schema and the application domain model.

Step 3: The extracted legacy schema and the mapping rules provide the input to the wrapper generator, which produces the source wrapper. The source wrapper at run-time translates queries from the application domain model to the legacy source schema.

In this thesis we focus exclusively on the build-time component which is described in Step 1. Specifically, we focus on developing robust and extensible techniques to extract application-specific meanings from code written for a legacy database. To get a comprehensive understanding of the source data and how it is represented, we augment

the schema information extracted by the SE with application-specific meanings discovered by the SA. Let us briefly illustrate our approach using a very simple example from the *Construction Scheduling* domain. Let us assume that when we extract the schema from the existing legacy source database using SE, we find that it contains a table named *Proj* which has an attribute named *p_id*. Without any further knowledge, we would probably not be able to understand the full meaning of these two schema items. This is where semantic analysis comes in. Let us assume we also have access to the application code that uses the database and upon further investigation, SA encounters output statements of the form: "Please Enter Project Number." Using program comprehension techniques described later in this thesis, SA can trace this and other output statements back to the corresponding items in the database where the values are eventually stored. So, if SA can trace the output statement above to *p_id* of *Proj* table, we can conclude that *Proj* stores information about projects and that *p_id* contains project numbers. Describing the details of how SA works is the focus of this thesis. Specifically, the research describe here extends some of the heuristics for semantic extraction developed previously [2] and produces a more robust, extensible infrastructure for semantic extraction with improved accuracy and performance.

1.3 Challenges

Semantic analysis can be defined as the application of techniques to one or more source code files, to extract semantic information to provide a complete understanding of the enterprise's business model. There are numerous challenges in the process of extracting semantic information from source code files with respect to the objectives of SEEK, which include the following:

- The functionality of the enterprise information system could be separated across several files. Thus, careful integration of the semantic information should be achieved.

- To gather information from multiple files, first the location of the file in the *classpath* or inside the *imported packages* should be specified.

- The SA may not have access or permissions to all the source code files. The accuracy and the correctness of the semantic information generated should not be affected even if partial or incomplete semantic information is extracted.

- High-level object oriented language features such as inheritance and method overloading increases the complexity of the SA.

- Source code and the underlying database design could be modified because of the changing business needs. Attributes with non-descriptive, even misleading names may be added to relations. The SA should correctly discover the application-specific meanings behind this attributes.

- Existing source code of a legacy system may not reflect the functionality of the running application. The source code might have been displaced or the version information of it may not be accurate. A way of using an application's binary code should be determined to get the latest functionality of the system.

- The semantic analyzer must be robust and must be capable of analyzing any source code of the target language.

- The semantic analyzer could be easily configured to accept other language source codes as an input.

- The semantic analysis approach should be generic enough to accept implementation of new heuristics to extract more information from the source code.

- Semantic analyzer should be easily maintained when an improvement is desired to be made on the existing functionality.

- Although the time performance is not a primary concern, SA should finish the analysis in a reasonable time period.

- SA should be able to parse any SQL statement, obtaining information from it.

1.4 Contributions

SA has brought the following contributions, which correspond to the challenges listed above:

- SA prototype can accurately integrate the information gathered from multiple files.
- Current SA can correctly find the location of the desired file in the *classpath* or inside the *imported packages* even if it is inside a compressed *jar* or *zip* files.
- The output of current SA still provides potentially valuable information for schema matching.
- High level object oriented language features like method overloading are successfully handled by SA.
- SA can correctly discover the application specific meanings of attributes even if the attributes names are non-descriptive or misleading.
- SA uses state-of-the-art code reverse engineering techniques to get the source code back from the binary code if the source code is misplaced.
- SA can parse and analyze any source code of Java language.
- SA can be easily extended to analyze other language source code inputs.
- SA can be extended to cover new heuristics.
- SA uses dynamic link list structures not to limit its capacity.
- SA's object oriented structure provides abstraction and encapsulation of the functionality which makes the maintenance easier.
- SA's time complexity has significantly decreased. It extracts the semantics with the same accuracy in a shorter period of time when compared to the previous version.
- Current version of SA can not only extract meaning from SQL *Select* statements, but also extract semantics from *Update, Insert,* and *Delete* statements.

1.5 Organization of Thesis

Chapter 2 presents an overview of the related research in the field of business rules, abstract syntax tree generator tools and design patterns. Chapter 3 provides a description of the approach for extracting semantics from the legacy code. Chapter 4 is dedicated to describing the implementation details of semantic analyzer. Chapter 5 highlights the power of the SA in terms of what kind of semantics it can capture. Finally, Chapter 6

concludes the thesis with a summary of our accomplishments as well as issues to be considered by future research projects.

CHAPTER 2
RELATED RESEARCH

In this chapter, we introduce the related research topics of our semantic analyzer. We have combined and used various techniques from different research areas to produce a novel methodology for extracting semantic information from legacy information sources.

The research areas mainly related with our semantic analyzer are *code reverse engineering, parser generation, design patterns, business rule extraction*, and *state-of-the-art knowledge extraction techniques* like *program slicing* and *pattern matching*. We not only base our work on these research areas, but also use state-of-the-art tools that are outcomes of these research areas as well. We introduce these research areas and tools in the order they are used in the implementation of semantic analyzer component of SEEK prototype. First, a code reverse engineering tool, *Java decompiler*, is introduced. Second, a parser generator tool, *Java Compiler Compiler* (JavaCC), is introduced. Introduction of the *Java Tree Builder* (JTB) tool, which is highly related with JavaCC, follows this section. Next, the *Visitor Design Pattern*, which has an important role in the structure of our approach, is introduced. The Section about lexical and syntactic analysis, control flow graph, program slicing and pattern matching techniques that are use in extracting semantic information follows. Last, we introduce the concept of business rules and the possible locations that they could be found in an information system.

2.1 Code Reverse Engineering

Source code is the input to our semantic analyzer. This is useful in situations where source code is misplaced or we can not be sure which source code was used to produce the running code in the legacy system. In such a case, we need a way to translate the running code back into source code by using code reverse engineering techniques. This section introduces the decompiler concept and the tool that we use to get *.java* source code files from *.class* files.

2.1.1 What is a Decompiler?

A compiler translates source code into machine code which the computer can interpret as actions or steps that the programmer wants it to perform

[1]. In this translation process, a lexical analyzer tokenizes the source code. The tokens are then passed to the language parser which matches one or more tokens to a series of rules and translates these tokens into intermediate code or into machine code. Any source code that does not match a compiler's rules is rejected.

Most compilers perform much pre- and post-processing. The pre-processor prepares the source code for the lexical analysis by omitting all unnecessary information, such as the programmer's comments and adding in any standard or included header files or packages. A typical post-processor stage is code optimization, where the compiler parses or scans the code, reorders it, and removes any redundancies to increase the efficiency and speed of the code.

[1] "Decompiling Java" Book by Godfrey Nolan, McGraw-Hill, 1998, 358 pages.

A *decompiler* translates the machine code or intermediate code back into source code. This transformation rarely results in original source code because of the pre- and post-processing stages.

2.1.2 History of Decompilers

The earliest example of a decompiler was written for the Algol-like language Neliac ("Navy Electronics Laboratory International ALGOL Compiler") in 1960. Its primary function was to convert non-Neliac compiled programs into Neliac compatible binaries. The first real Java decompiler was written at IBM by Daniel Ford[2]. Mocha, the most famous Java decompiler was written by Hanpeter Van Vliet[3].

2.1.3 Why Successful Decompiling is Possible in Java

Decompiling Java code requires a deep understanding of the Java byte code and the Java Virtual Machine (JVM) The current design of the Java Virtual Machine is independent of the Java Development Kit or JDK. It means that the language may change but the JVM specification is fixed. Decompilation can be done because the JVM is a simple stack machine with no registers and a limited number of high-level instructions or byte codes, making it easy to understand. The JVM requires the compiler to pass on much information such as variable and method names, which would otherwise not be available. The JVM's restricted execution environment and straightforward architecture as well as the high-level nature of many of its instructions help decompilers.

---

[2] Ford D, 'Jive: A Java Decompiler', at IBM, May 1996.

[3] Hanpeter V, C|Net article on August 23, 1996.

Adding a new method in C++ means that all classes that reference that class need to be recompiled. This is known as the fragile superclass problem[4]. Java gets around this problem by putting all the necessary symbolic information into the class file. The JVM then takes care of all the linking and final name resolution, loading all the required classes - including any externally referenced fields and methods - on the fly. This delayed linking or dynamic loading, possibly more than anything else, is why Java is so amenable decompilation.

2.1.4 Purpose of Decompilation

Decompilation is often used for retrieving lost source code, learning Java, understanding someone's algorithms or techniques, fixing and debugging class-files, and pretty printing. Our reason for Java decompilation is to get the *.java* source code from our own *.class* files, when we are not sure about whether *.java* source code files have the latest functionality of the legacy system.

2.1.5 Examples of Decompilers

There are several examples for decompilers. Here, we briefly introduce the first Java decompiler, Mocha, as well as the Jad decompiler that we have used in our project to decompile class files.

2.1.5.1 Mocha

Mocha is the first publicly available decompiler which was written by Hanpeter Van Vliet. He also wrote an obfuscator[5] called Crema that attempted to protect Java source code. The source code for both Crema and Mocha were sold to Borland. After

---

[4] Definition of the *Fragile Superclass Problem* can be found in the white paper "The Java Language Environment" by James Gosling and Henry McGilton, May 1996.

[5] An Obfuscator removes all clues that might have given by programmer when naming variables, methods, and parameters by altering the names into unintelligible names or even into control characters.

that, Borland released *JBuilder* with a built in obfuscator. Borland claims that JBuilder's obfuscator safeguards Java code from decompilation by replacing ASCII variable names with control characters. JBuilder comes with an API decompiler, which shows the public methods and their parameters. The same thing can be achieved using the *javap*[6] command in JDK.

2.1.5.2 Jad

Jad is the decompiler that we selected from among a number of decompilers because of its features [4]:

- Jad is a 100% pure C++ program and it works several times faster than other decompilers written in Java.
- Jad does not use the Java runtime for its functioning; therefore no special setup, like changes to the classpath variable, is required.
- Jad gives full support for inner and anonymous classes.
- Jad automatically converts identifiers garbled by Java obfuscators into valid ones.
- Jad is free for non-commercial use.

Jad can be used for the following reasons:

- For recovering lost source codes;
- For exploring the sources of Java runtime libraries;
- As a Java dis-assembler;
- As a Java source code cleaner and beautifier.

Like every decompiler, Jad has some limitations. While decompiling classes which contain inner classes, Jad cannot reliably sort out the extra arguments added to class constructors by Java compiler. Currently Jad makes no use of the Java class hierarchy information. Consequently, Jad always chooses *java.lang.Object* as a common super class of two different classes and inserts auxiliary casts where necessary.

2.2 Parser Generation

For programmers, writing a parser for a grammar has always been an error-prone task. Especially when we deal with a grammar like Java that has so many productions, it is also a time-consuming task to get a parser for that grammar. Therefore, there has been a lot of research on automatically generating a compiler from a formal specification of a

[6] javap: Java profiler outputs ASCII representations of a class file's byte code

programming language. Our focus in this thesis is extracting valuable information from legacy source code. For this reason, we choose to use one of this state-of-the-art parser generation tools to get our Java and SQL parsers, instead of writing our parsers from scratch. In this section, *Java Compiler Compiler* (JavaCC) tool, which is used to generate our parsers, is introduced.

2.2.1 Compiler-Compiler Tools

Research on automatic generation of compiler from a formal specification of a programming language brought us so-called *compiler-compiler* tools such as Yacc, Bison, ANTLR, and JavaCC. These tools play a key role in the software development process of compilers and have made their construction much easier.

Some of these well-known compiler-compiler tools are:

- **Lex & Yacc:** Lex builds lexical analyzers from regular expressions. Yacc converts a grammar specification into a table-driven compiler that could produce code when it had successfully parsed productions from that grammar.
- **Flex & Bison:** They are Free Software Foundation's GNU project produced improved versions of lex and yacc for use on platforms that did not run a derivative of the Unix operating system.
- **DFA and ANTLR** (Components of PCCTS[7]): They provide the same functions as Lex and Yacc. However, the grammars that ANTLR accepts are LL(k) grammars as opposed to the LALR grammars used by Yacc.

2.2.2 JavaCC and its Features

JavaCC (Java Compiler Compiler) is the most popular parser generator for use with Java applications [5]. It is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar according to that specification.

[7] PCCTS: The Purdue Compiler Construction Tool Set was developed at Purdue University.

JavaCC was first released by Sun with the name JACK. The latest version was released by Metamata Inc. WebGain[8] was the caretaker company of it for a while. Currently, JavaCC can be obtained from Sun Microsystem's Web site[9].

In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree), actions, debugging, etc. JavaCC does not have any runtime libraries (such as JAR files). The only files that are needed for the parser are those generated by JavaCC.

The parser, which is generated by JavaCC, is a top-down LL(1) parser[10] with a variable amount of look-ahead. JavaCC gives the ability to define LL(k[11]) grammars as well. Lexical specifications can define tokens not to be case sensitive either at the global level for the entire lexical specification or on an individual lexical specification basis.

The lexical analyzer of the JavaCC-generated parser has a feature for keeping comments when they are defined as special tokens. Special tokens are ignored during parsing, but these tokens are available (unlike skipped items, such as white spaces) for processing by the tools. In our implementation, we used this feature to gather comments associated with business rules and methods. Figure 2-1 below illustrates how special tokens are handled in a JavaCC parser. In Figure 2-1, *S1*, *S2*, and *S3* devote special tokens, while *T1, T2,* and *T3* devote the real tokens. If there were not a feature for keeping special tokens, *S1, S2*, and *S3* would be skipped during parsing the code. As shown in Figure 2-1, Token class has a *specialToken* member to point the special token

[8] http://www.webgain.com

[9] http://javacc.dev.java.net/

[10] Definitions of basic compiler terms can be found in Appendix B

[11] k: number of look-ahead.

that resides just before itself inside the code. T2 token, which is "*class"*, has two special tokens (comments) (S1:*"//open to public"* and S2:*"/* special token */"*) that are connected to its specialToken pointer. The left-hand side of Figure 2-1, is the figurative representation of the code that is on the right-hand side.

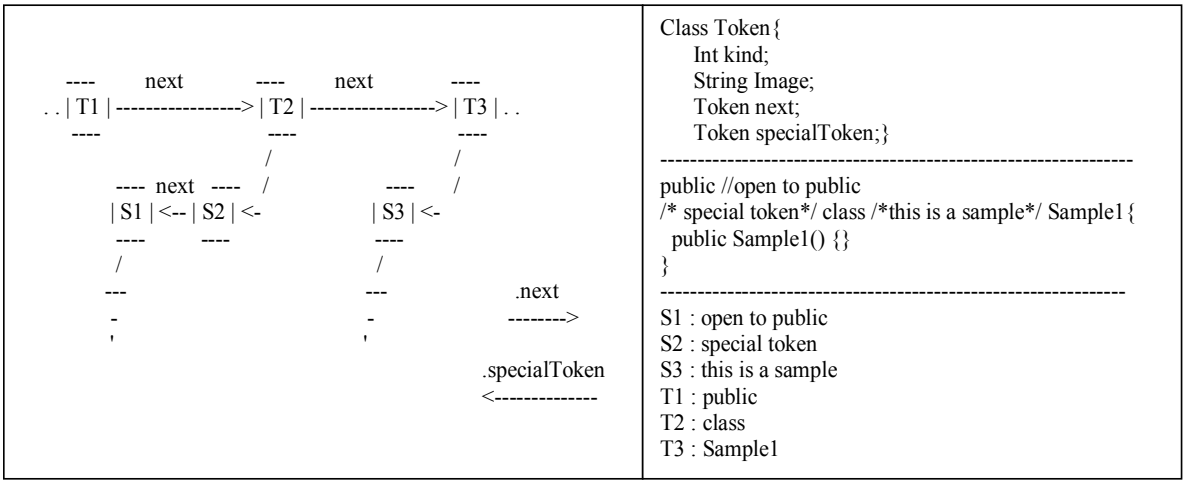


Figure 2-1 Special tokens in JavaCC

As shown in Figure 2-2, JavaCC processes specification file (.jj extension) and output the Java files that has the code of the parser that can process the languages that are according to the grammar in the specification file. Parser Java Files are both listed in Figure 2-2 and their functionality is described in Section 4.3.

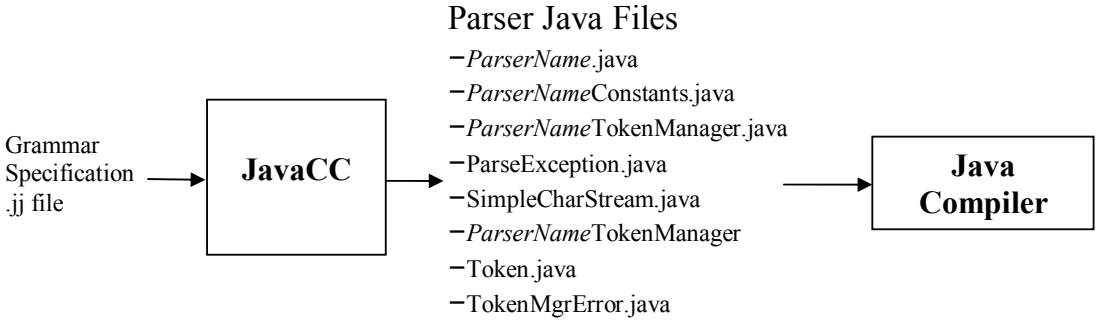


Figure 2-2 Input and output of JavaCC

Once the parser code is generated, it is compiled with the Java compiler (*javac*) to get the parser. The outcome code is used to parse the desired grammar, see Figure 2-3.

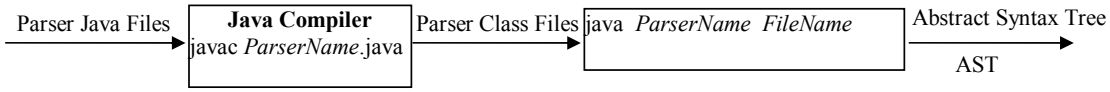


Figure 2-3 Compiling the Parser Generated by JavaCC and Using It to Get AST.

2.2.3 The Grammar Specification File of JavaCC

There are three major components in the grammar specification file.

- *Definitions of terminal tokens* (or terminals), e.g., words that are expected in the input.
- *Definitions of production rules* (or productions), e.g., rules about how tokens combine to form legal statements in the language.
- *Java code* to tie everything together and perform actions when certain productions occur.

Grammar specification files[12] for some grammars such as Java, C ++, C, SQL, XML, HTML, Visual Basic, and XQuery can be found at the JavaCC grammar repository Web site at UCLA [6].

2.2.4 Java Tree Builder (JTB)

In this subsection Java Tree Builder tool is introduced. We build our semantic analyzer structure according to the visitor pattern technique. Rationale behind the usage of visitor patterns is explained in the following section and in Chapter 3. JTB essentially takes a grammar specification file as an input and changes this specification file so that JavaCC can use this specification file to generate parsers according to visitor pattern structure. JTB, written at Purdue University, is used with the JavaCC parser generator tool [7]. It takes a plain JavaCC grammar file as input and automatically generates the following:

[12] An example of a grammar specification file can be found in Appendix A

- *A JavaCC grammar* with the proper annotations building the syntax tree during parsing. (The file jtb.out.jj)
- *A set of syntax tree classes* based on the productions in the grammar, utilizing the Visitor design pattern. (Files in syntaxtree directory/package)
- *Visitor and ObjectVisitor interface*, *DepthFirstVisitor* and *ObjectDepthFirst* visitors (Files in visitor directory/package)

The inputs and outputs of JTB are shown in Figure 2-4. The *syntaxtree* and *visitor* package outputs (Figure 2-4) are explained in the following paragraphs.

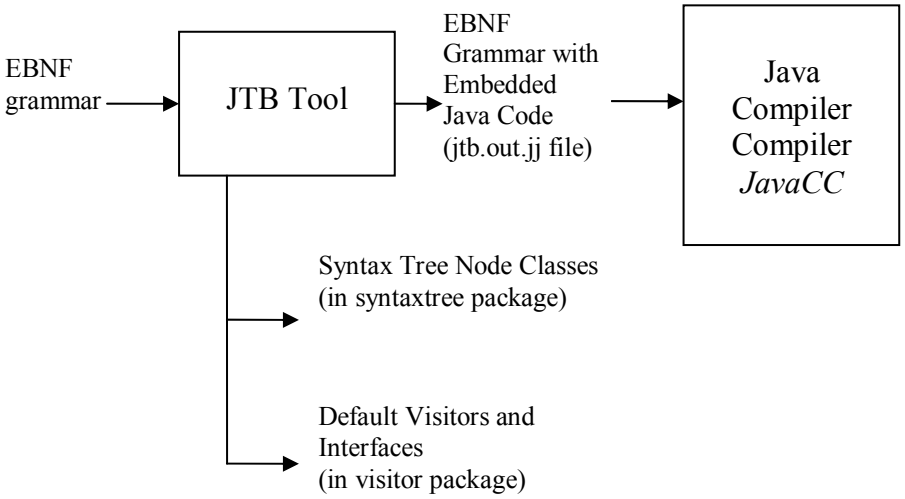


Figure 2-4: Inputs and Outputs of JTB Tool.

The *syntaxtree* directory/package contains syntax tree node classes generated based on the production rules in JavaCC grammar. Each production has its own class. Syntax directory also contains eight special node classes like “Node”, “NodeChoice”, “NodeOptional”, etc. Table 2-1 below specifies a list of these special nodes.

Table 2-1 Automatically generated tree node interface and classes

Node	Node interface that all tree nodes implement.
NodeListInterface	List interface that NodeList, NodeListOptional, and NodeSeqeunce implement.
NodeChoice	Represents a grammar choice such as (A | B)
NodeList	Represents a list such as (A)+
NodeListOptional	Represents an optional list such as (A)*
NodeOptional	Represents an optional such as [A] or (A)?
NodeSequence	Represents a nested sequence of nodes
NodeToken	Represents a token string such as "package"

The generated classes include accept methods with the visitor interface as a formal parameter. This parameter is used to pass the name of the class back to the visitor through dynamic binding. Each *accept* method takes a visitor as an actual parameter and invokes the visit method of that visitor having the node class as an actual parameter to determine the appropriate visit method. [8]

Separating the generated tree node classes into their own package greatly simplifies file organization. When the grammar is stable and not subject to change, once these classes are generated and compiled, it is not necessary to deal with them again. All of the changes and additions are done to the visitor classes when a new functionality is introduced or an existing functionality is changed.

The *visitor* directory/package contains the visitor interfaces, *Visitor* and *ObjectVisitor*, and their default implementations, *DepthFirstVisitor* and *ObjectDepthFirst*.[13]

The Visitor interface contains one visit method declaration per production in the grammar, plus one visit method declaration for each of the automatically generated special node classes. The default visitor, *DepthFirstVisitor*, class implements the visitor interface and it simply visits each node of the tree. The JTB puts the navigation control into the visit methods of the Visitor [9]. JTB generates default navigation behavior in the implementation of the default visitor, *DepthFirstVisitor* or *ObjectDepthFirst*. A visitor that wants to use this route to process operations on specific nodes of tree extends the default visitor and overrides the method related to those specific nodes[14].

---

[13] An example of visitor class implementation can be seen in Appendix D.

[14] An example of a visitor class can be found in the Appendix D.

Operations on abstract syntax trees can then be defined separately from the abstract syntax classes in visitor classes that rely on the generated accept methods. The separation of operation and structure is crucial in this class of tools to avoid editing and recompiling generated code [9].

2.3 Visitor Design Pattern

Our goal is to build semantic information extraction techniques that can be applied to any source code and can be extended with new algorithms. Visitor design pattern is the key object oriented technique to reach this goal. In this section, we introduce the Visitor Design Pattern and its features.

2.3.1 Design Patterns

Design Patterns [10] are general solutions to design problem occurring repeatedly in many projects. The Design Patterns can be listed in three categories:

- **Creational Patterns:** Abstract Factory, Builder, Factory Method, Prototype, and Singleton.
- **Structural Pattern:** Adapter, Bridge, Composite, Decorator, Flyweight, and Proxy.
- **Behavioral Patterns:** Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

2.3.2 Features of Visitor Design Pattern

A *Visitor design pattern* is a behavioral pattern which is used to encapsulate the functionality that is desired to be performed on the elements of a data structure. It gives the flexibility to change the operation being performed on a structure without the need to change the classes of the elements on which the operation is performed. Visitors make it easy to add operations. New operations over the object structure can be defined simply by adding a new visitor. In contrast, if the functionality is spread over many classes, each

class must be changed to define a new operation[15]. The idea behind a visitor pattern is to insert an *accept* method in each class. The accept method passes control back to the visitor which acts as a repository for the new methods[16]. Visitor classes localize related behavior in the same visitor and unrelated sets of behavior are partitioned in their own visitor subclasses.

Using visitor patterns, each new operation can be added separately and the member classes are independent of the operations that apply to them. If the classes defining the object structure rarely change, but new operations over the structure are often defined, a visitor design pattern is the perfect choice. Since object structure classes do not change, we do not need to recompile them when new functionality is added.

2.3.3 Analogy for Visitor Pattern

It is often difficult to understand design patterns. To explain concepts like design patterns, analogies from real life are useful. To increase the reader's understanding of visitor pattern, we introduce the following analogy.

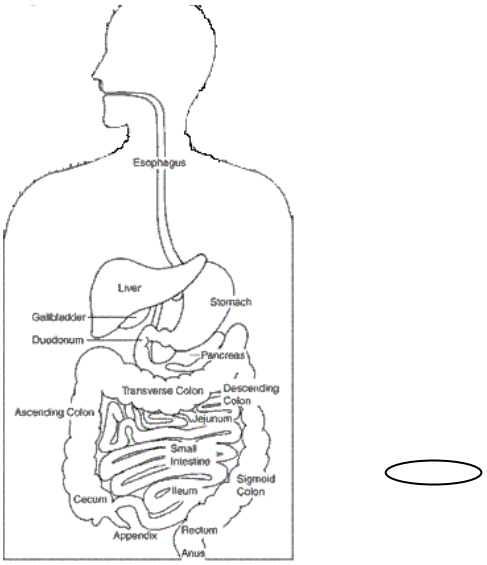
Let us suppose that our concrete, invariant structure is the digestive system of human-body and our visitor is a computerized pill. Let us further assume, we would like to get information from our digestive system by using the pill as an information carrier. When the pill is swallowed, it visits the organs of our digestive system and sends information about those organs to the outside world. In this way, no surgery is needed to put devices inside each organ to send information to the outside world. The pill sends information about the organs while it is traversing through the digestive system. The

---

[15] Comparison of possible approaches (*Instanceof* and *Type Casts*, *Dedicated Methods* and *The Visitor Pattern*) for performing operations on node objects in a heterogeneous aggregate structure are discussed in detail in [5].

[16] An example of a visitor pattern implementation can be found in Appendix D.

information that is send is processed in the outside world. Figure 2-5 may help you to imagine this analogy.



A B

Figure 2-5 Analogy for visitor pattern. A) Human digestive system (Concrete structure) B) Pill (Visitor Structure).

2.3.4 Double Dispatching in Visitor Patterns

The visitor pattern mechanism provides double dispatching for the selection of the functionality that will be performed on a specific node. This means that the call made to the method that inherits the functionality for a specific node depends both upon the type of the visitor and of the data structure of the node. This is achieved as follows: Each node in the data structure has an *accept* method that deals with the abstract visitor. When a node accepts a visitor, it makes a call to the visitor which includes the node's class. The visitor will then execute its algorithm for that element. The double dispatching technique completely replaces the need for conditional statements.

2.4 State-of-the-art in Extracting Knowledge from Application Code

In this section, we introduce several techniques for extracting information from source code. The problem of extracting knowledge from application code has been investigated in several research areas such as lexical and syntactic analysis, control flow analysis, program slicing, and pattern matching. These areas are described in the previous work about semantic analysis applied to SEEK project [2]. We briefly describe these techniques, since they are used by our semantic analysis approach.

2.4.1 Lexical and Syntactic Analysis

Analysis of source code requires conversion to a form that can easily be processed by a program. An abstract Syntax Tree (AST) is a type of representation of source code that facilitates the usage of tree traversal algorithms. Lexical and syntactic analysis of a source code, in accordance to the context free grammar of the underlying language, generates an AST. Grammars themselves are described in a stylized notation called *Backus-Naur Form* [11] in which the program parts are defined by rules and in terms of their constituents. An AST is used to show how a natural language sentence is broken up into its constituents. The parser that is generated by JavaCC produces AST of a source code. After the AST is produced, it is ready to be analyzed by visitor patterns.

2.4.2 Control Flow Analysis

After we obtain the AST, it is used for Control Flow Analysis [12]. There are two major types of CFA; *Inter-procedural* and *Intra-procedural* analysis. Inter-procedural analysis determines the calling relationship among program units while intra-procedural analysis determines the order in which statements are executed within these program units. Together they construct a Control Flow Graph (CFG). In our approach we

construct the CFG and use the calling relationships on CFG between methods to traverse between classes.

2.4.3 Program Slicing

Program slicing was introduced by Weiser [13] and has supported various program comprehension techniques. Weiser defines the slice of a program for a particular variable at a particular line in the source code as "that part of the code that is responsible for giving a value to the variable at that point in the code". The idea behind slicing is to retrieve the code segment that has a direct impact on variables of interest. Starting at a given point in the program, program slicing automatically retrieves all relevant code statements containing control and/or data flow dependencies.

2.4.4 Pattern Matching

Pattern matching identifies interesting code patterns and their dependencies. For example, conditional control structures such as *if..then..else* or *case* statements may encode business knowledge, whereas data type declarations and class or structure definitions can provide valuable information about the names, data types, and structure of concepts represented in an underlying database. A code pattern recognizer performs the matching of patterns from the templates against patterns in the source code.

2.5 Business Rules Extraction

In this last section of Chapter 2, we investigate the research on business rules. Extracting business rules embedded in information systems is an important issue in understanding the functionality of the information system. Before attempting to solve the problem of extracting business rules, we define what a business rule is, what kinds of business rules typically exist, and how they are encoded in the information systems.

2.5.1 Business Rules

Business rules represent core business practices and policies, and they determine what is possible or desirable in running a business [14]. A well-known definition states that: “A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business [15]. “

2.5.2 Categories of Business Rules

Business rule may be implemented differently in different parts of a system and require different techniques to extract them. For this reason, it is beneficial to know the categories of business rules and how they could be implemented in an information system. For example [14], [15] provide two possible ways to categorize business rules. Using the approach suggested in [14], we can distinguish the following classes of business rules:

1. Structural business rules: Structural business rules are statements about the data objects belonging to an organization. Structural business rules can be further divided into following categories:

- Definitions

Example: *Anyone placing an order with us is a customer.*

- Relations between objects

Example: *Customers are identified by their address, postcode, and contact.*

- Derivation statements

Example: *The total cost of the rental is calculated from the sum of insurance amount and rental amount.*

2. Behavioral business rules: Behavioral business rules are statements about dynamic aspects of a business. They specify what can or will be done to data objects in response to events. The actions taken in response to the events may result in a state change for the data objects affected by the business rules.

Example: *When an insurance claim is made, a claim reference number is allocated.*

3. Constraint business rules: Constraint business rules are statements that bring limitations on the operation of the business rules. Constraint business rules define the conditions under which a business can operate.

Example: *A customer must have at least one account, but no more than three.*

2.5.3 Where Are Business Rules Implemented in Information Systems

Categories of business rules that are stated in the previous section can be implemented in various parts of an information system. It is beneficial to identify what we are looking for before we look for it. If we know what kind of business rule we can find in the subparts of the information system, then we can construct our search algorithms according to the category of the business rule for which we are looking. Therefore, we define which business rule categories could be implemented in which part of the information system.

Structural business rules: Definitions which are also a part of structural business rules are trying to be supported by the ontology. Relations which are also structural business rules are generally defined by the data model and implemented in the schema. Derivation business rules are implemented as program code.

Behavioral business rules: Behavioral business rules often implement more complex business logic than can be defined through a data model, and they are usually implemented as procedural programs.

Constraint business rules: Constraint business rules may be implemented in various parts of an information system. In older systems, constraint business rules tend to be implemented as program code. In newer systems, they are also implemented in the schema.

2.5.4 Extracting Business Rules

The aim of extracting business rules is to understand the functionality of information system. There are mainly four inputs to our methodology for business rule extraction. These are:

- **Schema:** It is the main input for data understanding, and it provides structural and some constraint information about data.
- **Data:** The data stored in a database may also be examined for data understanding purposes. This is because data may contain certain properties that are not specified in the schema. Also business rules that were extracted from data can be used to verify the business rules that were extracted from other inputs (namely, schema or programs).
- **Programs:** Program code associated with a database system is another type of input for data understanding. A database system may include application programs, stored procedures, or user interface forms. These programs contain clues about data structures and constraints. These clues, when used in conjunction with schema/data, could potentially increase data understanding.
- **Transactions:** Operations are recorded in a transaction log in a database system. A transaction log may be used as an input for data understanding. The sequence of operations on a database, and the relationship among operations that take place in one transaction, can be beneficial to understand the functionality of the information system.

2.5.5 Importance of Business Rule Extraction from Program Code

Over time, business rules evolve and the software that encodes them is also changed and maintained. As the information system becomes large and ages, the changes in business functionality are reflected in program code, without changing the corresponding documentation, or without changing any other part of the information system. Thus, the business organization often trusts the code more than any other documents. Therefore, extracting business rules from program code becomes more important for a better understanding of the legacy information systems. Information gathered from the business rules inside the program code are then used to semantically enhance the schema of the information system.

CHAPTER 3
SEMANTIC ANALYSIS ALGORITHM

3.1 Introduction

A conceptual overview of the SEEK knowledge extraction architecture is shown in Figure 3-1 [1]. SEEK applies Data Reverse Engineering (DRE) and Schema Matching (SM) processes to legacy database(s), to produce a source wrapper for a legacy source. This source wrapper will be used by another component (not shown in Figure 3-1) to communicate and exchange information with the legacy source. It is assumed that the legacy source uses a database management system for storing and managing its enterprise data or knowledge.

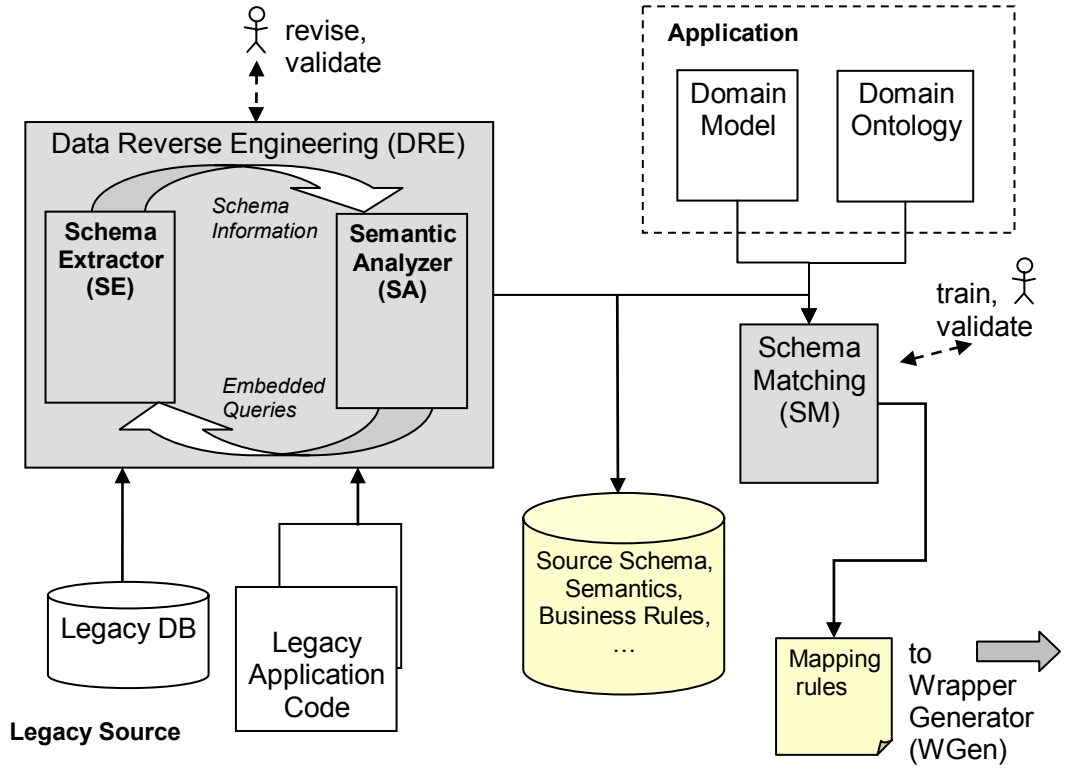


Figure 3-1 High-level view of the SEEK Architecture.

First, SEEK generates a detailed description of the legacy source by extracting enterprise knowledge from it. The extracted enterprise knowledge forms a knowledge base that serves as the input for subsequent steps. In particular, the DRE module shown in Figure 3-1 connects to the underlying DBMS to extract schema information (most data sources support at least some form of Call-Level Interface such as JDBC). The schema information from the database is semantically enhanced using clues extracted by the semantic analyzer from available application code, business reports and, in the future, perhaps other electronically available information that could encode business data such as e-mail correspondence, corporate memos, etc. It has been the experience, through visits with representatives from the construction and manufacturing domains, that such application code exists and can be made available electronically [1].

Second, the semantically enhanced legacy source schema must be mapped into the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema mapping process that produces the mapping rules between the legacy source schema and the application domain model. In addition to the domain model, the schema matcher also needs access to the domain ontology (DO) that describes the domain model. Finally, the extracted legacy schema and the mapping rules provide the input to the wrapper generator (not shown), which produces the source wrapper.

The three preceding steps can be formalized as follows [1]. At a high level, let a legacy source L be denoted by the tuple $L = (DB_L, S_L, D_L, Q_L)$, where DB_L denotes the legacy database, S_L denotes its schema, D_L the data, and Q_L a set of queries that can be answered by DB_L. Note, the legacy database need not be a relational database, but can

include text, flat file databases, or hierarchically formatted information. S_L is expressed by the data model DM_L.

We also define an application via the tuple $A = (S_A, Q_A, D_A)$, where S_A denotes the schema used by the application and Q_A denotes a collection of queries written against that schema. The symbol D_A denotes data that is expressed in the context of the application. We assume that the application schema is described by a domain model and its corresponding ontology (as shown in Figure 3-1). For simplicity, we further assume that the application query format is specific to a given application domain but invariant across legacy sources for that domain. Let a legacy source wrapper W be comprised of a query transformation (3-1) and a data transformation (3-2) where the Q and D are constrained by the corresponding schemas.

$$f_W^Q : Q_A \rightarrow Q_L \quad (3\text{-}1)$$

$$f_W^D : D_L \rightarrow D_A \quad (3\text{-}2)$$

The SEEK knowledge extraction process shown in Figure 3-1 can now be stated as follows. Given S_A and Q_A for an application that attempts to access legacy database DB_L whose schema S_L is unknown, and assuming that we have access to the legacy database DB_L as well as to application code C_L that accesses DB_L, we first infer S_L by analyzing DB_L and C_L, then use S_L to infer a set of mapping rules M between S_L and S_A, are used by a wrapper generator WGen to produce (f_W^Q, f_W^D). In short:

$$\text{DRE: } (DB_L, C_L) \rightarrow S_L \quad (3\text{-}3)$$

$$\text{SM : } (S_L, S_A) \rightarrow M \quad (3\text{-}4)$$

$$\text{WGen: } (Q_A, M) \rightarrow (f_W^Q, f_W^D) \quad (3\text{-}5)$$

Thus, the DRE algorithm (3-1) is comprised of schema extraction (SE) and semantic analysis (SA). This thesis concentrates on the semantic analysis process by analyzing application code C_L to provide vital clues for inferring S_L. The implementation and experimental evaluation of the DRE algorithm have been described in detail in [1]. The following section focuses on the semantic analyzer approach.

3.2 Objective of Semantic Analysis

The main purpose of semantic analysis is to understand the meaning of database repository. In the process of semantic analysis, we extract information that can be used to enhance the schema of the legacy source (S_L). This S_L will then be used to produce mapping rules between S_L and the applications (S_A) that want to access the legacy source. Our input to the semantic analysis process can be any kind of code (C_L) that inherits functionality that operates on the legacy database (DB_L). This code can be application code, as well as triggers or stored procedures inside the corresponding database. Our current prototype works on application code and extracts semantics out of that by using state-of-the-art techniques in code reverse engineering, abstract syntax tree generation and object oriented programming. Our prototype is easily extensible to analyze any programming language, trigger codes, and stored-procedure code that exist inside legacy databases.

Our current semantic analyzer is configured to extract information from Java code. We choose the Java programming language because it is becoming the dominating programming language in the enterprise information systems. Also, Java promises to be the programming language that inherits semantics of future legacy systems. In the next

section you will find detailed explanation of the main components of our Java semantic analyzer.

3.3 Components of the Semantic Analyzer

The semantic analyzer (Figure 3-2) has three main components.

- Decompiler,
- Abstract syntax tree generator
- Information extractor.

Let us give a detailed description of the techniques and rationale used to form each of these components.

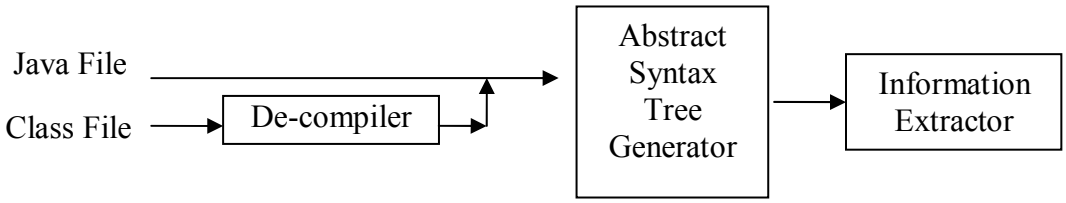


Figure 3-2 Semantic Analyzer Components.

3.3.1 Decompiler

Legacy information systems often have poor or outdate documentation. Application code is changed over time and documentation becomes inconsistent with the functionality of the code. In such a case, the most reliable source of semantic information of the legacy system becomes the application source code. There can be situations where source code could be misplaced, or the versions of source code could be disorganized. In that case, we can not be sure as to which source code used to produce the running code on the legacy system and which source code was has the latest functionality of the legacy system. In this situation, the only reliable information source may be the running application, *.class* files. We can recover Java source code (*.java)* files by the decompiler component of semantic analyzer. As it is illustrated in Figure 3-2, using decompiler is optional. If we

are sure that .j*ava* files reflect the latest functionality of the legacy system, we can skip the decompiler stage.

How can we be sure that the original source code and the de-compiled source code are the same? There is a way to test this. We can recompile the .java file of decompiled .class file with *"javac"* and check if the sizes of the original class file and the recompiled class files are the same. The approximation of the length of the sizes of these two files will give us a clue about how the decompiling process is successful. The Figure 3-3 illustrates this test.

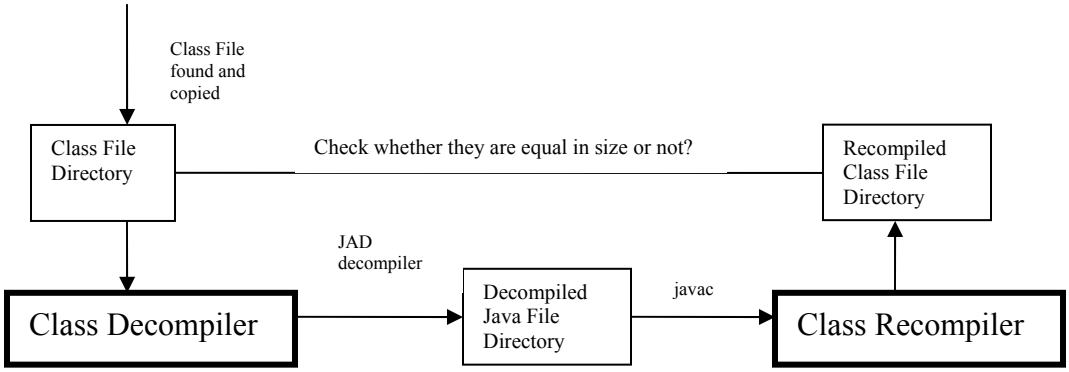


Figure 3-3 Checking the success of decompiling process.

3.3.2 Abstract Syntax Tree Generator

Our second component is the abstract syntax tree generator. The essence of the abstract syntax tree (AST) in program comprehension is introduced in Chapter 2. We use JavaCC tool to automatically generate AST generator programs. Detail description of JavaCC tool and the idea behind using this tool is given in Chapter 2. The AST generator (parser) that is produced by JavaCC was tested over thousands of java program code. This makes our AST generation process robust and error-free.

3.3.3 Information Extractor

The information extractor component, which is shown in Figure 3-2, is composed of several visitor classes. These visitor classes traverse on the AST of the application code to extract the desired information. A visitor pattern lets programmer to encapsulate related behavior in the same visitor and separate the unrelated ones into other visitors. Our information extractor contains visitor patterns to extract variable information, to identify the potential value of variables, to extract the flow of program code, to extract the business rules, to extract the comments inside the code, to extract the SQL statements, and to write the AST in an XML file. If we need any new operation over the object structure, it can be defined simply by adding a new visitor. Detailed descriptions and implementations of visitor patterns are explained in Chapter 2 and Chapter 4.

There are two main steps in the algorithm that extract semantic information. In the first step, control flow graph (CFG) is extracted, declaration information of each variable and method are written into symbol table and AST of the program code is written into an XML file. In the second step, we identify the slicing variables and get the business rules, comments, SQL statement, and relate meanings of the variables with the table columns. These steps are illustrated in Figure 3-4 and are explained in detail in Section 4.4.

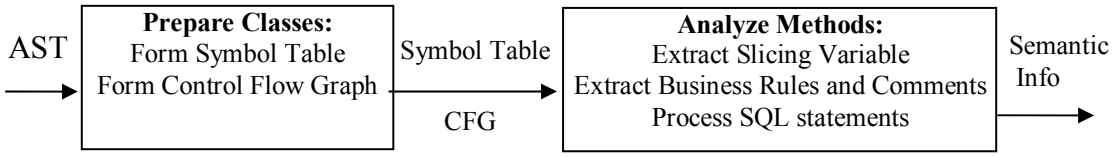


Figure 3-4 Two main steps of Information Extraction.

In the first step, information about all variables is gathered and the AST of each method is saved. In the second step, we do not analyze the class as whole but we analyze

only the methods in the class that are used by method calls. By this approach, we save work and improve performance.

3.4 Extensibility of the Semantic Analyzer

The objective of semantic analysis is to develop techniques for extracting information from legacy systems. The input of the semantic analysis process can be a trigger code, a stored procedure, or a java code. More generally, the input can be any code that expresses semantic information about the legacy source on which it operates. For this reason our techniques in semantic analysis should be extensible to any programming language platform.

Extensibility of the semantic analyzer is achieved because *JavaCC* provides an AST generator that can easily be plugged in and out. Abstraction of AST generator from the information extractor increases the extensibility of semantic analyzer. If the functionality of information extraction and AST generation are fully separated from each other, then the extensibility of semantic analyzer to other languages will be much easier. This separation of functionality of these two components is supported by visitor patterns. Visitor patterns provide the flexibility to change the operation being performed on a structure without the need for changing the classes of the elements on which the operation is performed. By using visitor patterns, we do not embed the functionality of the information extraction inside the classes of AST generator. This separation lets us to focus on the information extraction algorithms regardless of the programming language that is being analyzed. To get a structure for Visitor pattern usage we use the *JTB* tool [7] to augment our structure to be used by visitor patterns. A detailed description and implementation of JTB is given in Chapter 2 and Chapter 4.

3.5 Extracting Semantics from Multiple Files

The current version of the SA prototype can extract semantics from multiple class files by starting from an initial driver program file and analyzing all program files in the control flow graph. The functionality of typical enterprise applications spreads over multiple files. Therefore, one must extract semantics from multiple files and combine them for a better understanding of legacy source.

Figure 3-5 gives an example of how functionality could be separated into different files. In this example, we start from the main method of *Class1* source file. Inside the main method, *method2* of *Class2* class is invoked. This means, we need to examine *method2* of *Class2*. To examine it, our File Finder finds the location of the source code file of *Class2*. Whenever a method is invoked on another class, this class source file is found and process continues from this class file.

As we have mentioned before, there are two main steps in information extraction. In the first step, we find all the class files that will be analyzed and prepare the control flow graph by finding out inter- and intra-procedural relationships. In the second step, we analyze the methods in the order they are called starting from the *main* method of the driver class. These steps are explained in detail in Section 4.4.

It can be readily observed in Figure 3-5 that all the methods of a class file may not be used. For this reason, our information extractor only analyzes the methods that are used in the control flow. In the example Figure 3-5, we do not analyze *method1* of *Class1*, *method1* of *Class2*, and *method3* of *Class3*. Analyzing the code, method by method, helps us to improve not only the temporal performance of semantic analyzer, but also its correctness. Unused code (methods) inside classes may lead to incorrect, misleading information.

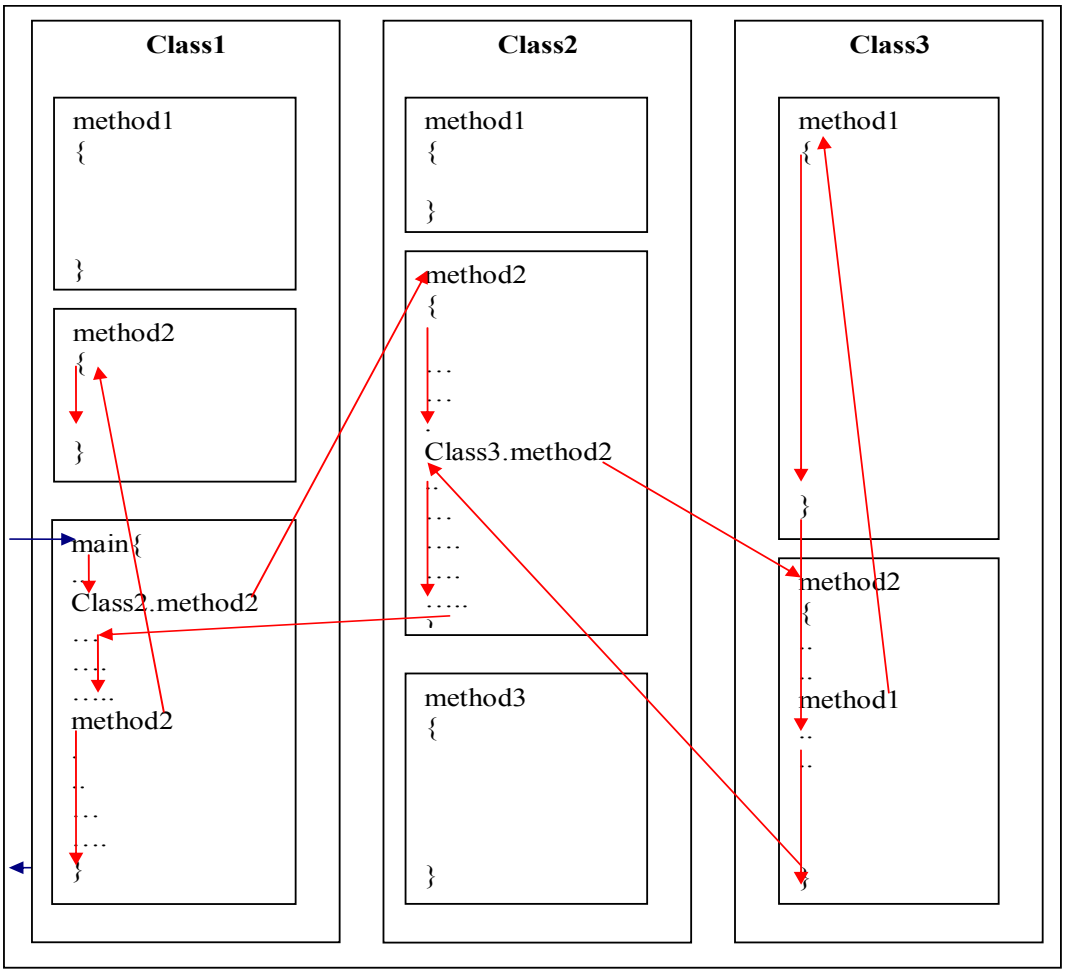


Figure 3-5 Traversing between class files

3.6 Finding the Class Source Code Files

The Semantic Analyzer follows the same logic as the Java Virtual Machine (JVM) to find a class source code file. When we run a Java application, the JVM first searches for the class that was given as a parameter. JVM searches for this class file in the directories that are stated in the environment variable *ClassPath*. When JVM finds the class file, it starts executing it from its main method. When a method is invoked on an instance of a class, JVM continues its execution inside that method of the specified class. JVM finds this class by using both the *ClassPath* information and package import information of the class that is currently being executed.

The File Finder module of the SA imitates the same logic to find the files that is analyzed. File Finder module first gets the *ClassPath* information. As we traverse through the classes, the current class's package/class import information is given to the File Finder. When a new class file is searched, File Finder uses both *ClassPath* and import information to find the location of the searched file. As it is illustrated in Figure 3-6, *ClassPath* information is initially given to File Finder module. While we traverse between classes, each class package and import information is feed backed to the File Finder. When a class is to be analyzed, its name is given to File Finder and its source code file location is found by the File Finder. File Finder can correctly find the location even if it is inside a compressed *jar* or *zip* file.

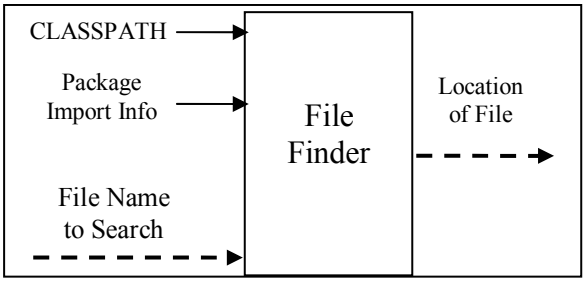


Figure 3-6 File Finder

3.6 Slicing Variables

The semantic analysis algorithm is based on several observations based on the general nature of legacy application code. Database application code always has embedded queries. The data retrieved or manipulated by queries is displayed to the end user. Both the queries and the output statements contain rich semantic information [2]. The SEEK SA aims at augmenting entities and attributes in the database schema with their application specific meanings. It is reasonable to state that the variables that appear in input/output or on database statements should be traced throughout the application code. We will call these variables *slicing variables* [2]. No matter how large the number

of slicing variables is, we can extract the semantics by a single pass through the AST. In each input-output statement we check if the variable in the statement is a slicing variable or not. If it is a slicing variable, we associate the meaning that was gathered thus far with that slicing variable. The single pass does not cause any information loss because of the data structure used, but it decreases the run-time complexity of the algorithm.

3.7 Heuristics Used

In this section, we describe the heuristics used in extracting information from input/output statements.

Heuristic1: Application code generally has input-output statements that display the results of queries executed on the underlying database.

Typically, output statements display one or more variables and/or contain one or more format strings. A format string is defined as a sequence of alphanumeric characters and escape characters within quotes. An escape character is a backslash character and followed by a sequence of alphanumeric characters (e.g., *\n, \t, \r* etc), which in combination indicate how to align and format the output.

System.out.println("\n Activity Name: \t" + v);

In the example above, the string *"\n Activity Name:"* in the statement represents the format string. The escape sequence *"\n"* specifies that the output should begin on a new line.

Heuristic 2: The escape characters in format string do not include any semantic meaning, so we can eliminate them. In the example below; *'\n'* and *'\t'* have no semantic meaning.

System.out.println("\n Activity Name: \t" + v);

Heuristic 3: The set of slicing variables includes variables that appear in input, output, or database statements. This is the set of variables that provide the maximum semantic knowledge about the underlying legacy database.

Heuristic 4: The format string in an input-output statement describes the displayed slicing variable that comes after this format string.

System.out.println("\n Activity Name: \t" + v);

In the example above, the semantic of the variable *v* exists inside the format string "*\n Activity Name: \t"*.

Heuristic 5: The format string that contains semantic information and the variable may not be in the same statement.

System.out.println("\n Activity Name:");

System.out.print(v);

Let us refer to the first output statement with the format string as *s1* and the second output statement that actually outputs the value of the variable as *s2*. Notice that, *s1* and *s2* can be separated by an arbitrary number of statements. The format string *s1* gives clues about the application meaning of *v* in output *s2*.

Heuristic 6: There may be an arbitrary number of format strings in different statements that inherits semantics, before we encounter an output of slicing variable. Concatenation of the format strings before the slicing variable will give more clues about the variable semantic.

System.out.println("\n Project:");

System.out.println("\n Activity Name:");

System.out.print(v);

In the example above, variable *v* is described by both *"\n Project:"* and *"\n Activity Name:"* format strings. Let us call the first output statement with the format string as *s1*, the second output statement with the format string as *s2*, and the second output statement that actually outputs the value of the variable *s3*. Notice that, *s1, s2,* and *s3* can be separated by an arbitrary number of statements. In such a case, all the format string should be concatenated to form the format string that gives clues about the application meaning of *v*.

Heuristic 7: There may not be any output statement that has format string before the output statement of variable. The semantic information of variable may be given before the output statement of another statement. In the example below, the semantic of variables v1, v2, and v3 are all inherited inside the format string of first output statement.

System.out.println("\n Project \t Activity \t Work Item");

System.out.print("\t" + v1);

System.out.print("\t" + v2);

System.out.print("\t" + v3);

A classic example of this situation in database application code is the set of statements that display the results of a SELECT query in a matrix format. The matrix title and the column headers contain important clues about the application specific meanings of variables displayed in the individual columns of the matrix.

Heuristic 8: If a statement *s* assigns a value to variable *v*, and *s* retrieves a value of a column *c* of table *t* from the result set of a query *q*, we can associate *v*'s semantics with column *c* of table *t*.

resultset1 = s.executeQuery("Select number from Activ");

no = resultset1.getString(1);

System.out.println("Activity no: " + no);

In the example above, "*no"* variable is associated with the semantic *"Activity no"*. It is also associated with first column of the query which is *"number"*. So the column *"number"* can be associated with the semantic *"Activity no"*.

Heuristic 9: If a variable is used in the where clause of a SQL statement, then this variable's application meaning can be associated with the column in the where clause.

query = "SELECT activ FROM Activity WHERE activ_no = '"+input+"'";

resultSet = sqlStatement.executeQuery(query);

System.out.println("Activity no: " + input);

The example above shows an example of how a variable can be used in the *where clause* of a SELECT condition. In this example, the *input* variable's meaning can be associated with the meaning of *activ_no* column. This heuristic can be used for UPDATE, DELETE, and INSERT conditions as well. The w*here clause* of any SQL statement can give clues of variable-column relations.

Heuristic 10: If a variable is used to set the value of a column in the UPDATE SQL statement, then this variable's application meaning can be associated with that column. In the example below, the application specific meaning of *name* variable can be associated with the column *active_name*.

updateSQL = "UPDATE Activity SET active_name = '"+name+"' WHERE activ_no='"+first + "'";

sqlStatement.executeUpdate(updateSQL);

Heuristic 11: Business logic is encoded either as assignment statements or conditional statements like *if..then..else*, *switch..case*, etc. or a combination of them. Mathematical formulae translate into assignment statements while decision logic translates into conditional statements [2].

If slicing variable *v* is part of an assignment statement *s* then statement s represents a business rule involving variable *v*. The assignment, in the below example, is a potential business rule.

```
slicingVar1 = var1 + var2;
```

Heuristic 12: If slicing variable *v* appears in the condition expression of an *if..then..else* or *switch..case* or any other conditional statement *s*, then *s* represents a business rule involving variable *v*. In the example below, variable *"NoOfPurchases"* is a slicing variable.

```
if (NoOfPurchases > 50){
        totalAmount = totalAmount * 0.9;
}
```

Heuristic 13: Another source of information could be comments inside the code. General policy of writing comment is to write them right before the statement with what it is related. We assume that comments about business rules, if written, exist just before the business rule. In example below, there is a comment right before the business rule.

```
// if number of purchases exceeds 50, make a 10% discount on the total prize
if (NoOfPurchases > 50){
totalAmount = totalAmount * 0.9;
}
```

Heuristic 14: If there is a comment about the functionality of a method, general policy of writing this comment is to write it right before the method declaration. We assume that comments about method functionality, if written, exist just before the method declaration. In the example below, the comment about the method is placed right before its declaration.

```
/**
* Method: void parseClassPath()
* parses CLASSPATH string and fills the classPart array
*/
public void parseClassPath()
{String tempPath = classPath;
```

CHAPTER 4
IMPLEMENTATION OF THE JAVA SEMANTIC ANALYZER

As illustrated in Figure 4-1, the Schema Extractor (SE) and Semantic Analyzer (SA) are embedded inside the data reverse engineering (DRE) module of the SEEK prototype. The SE connects to the underlying DBMS to extract schema information and has been implemented by my previous colleague in SEEK [1],[2]. The schema information from the database is semantically enhanced using clues extracted by the SA. We have integrated the current Java SA code with the SE code as well, but we do not describe the integration details herein. Instead, in this chapter, we focus on the implementation details of the current Java SA prototype. Legacy application code is the input of SA, whose output consists of semantic descriptions for the legacy source schema extracted by DRE. In addition, SA elicits business rules describing some of the enterprise knowledge that is implicitly embedded in the application code.

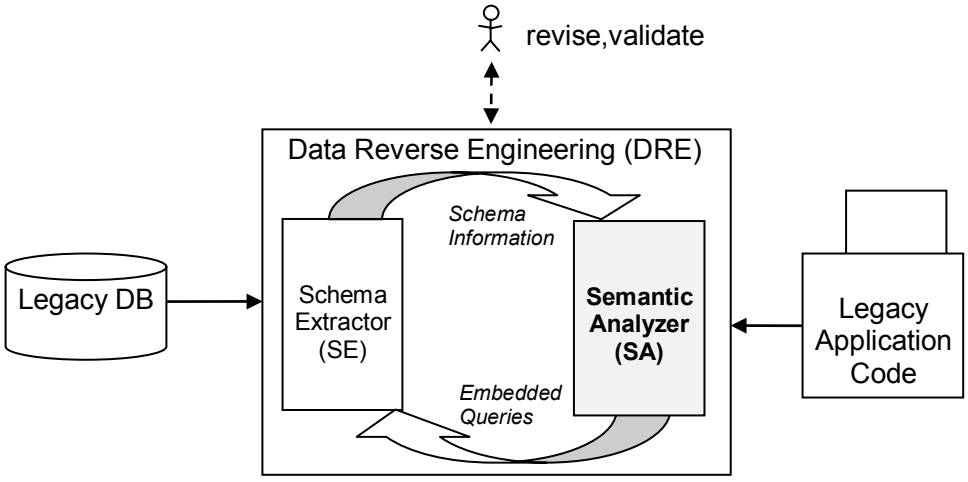


Figure 4-1 Architectural overview of the SA prototype and related components.

The SA prototype is implemented using the Java SDK 1.4 from Sun Microsystems. The Jad v1.5.8 decompiler is used to convert class files to Java source files. Java Tree Builder (JTB) v1.2.2 and JavaCC v1.1 are used to generate Java and SQL parsers.

4.1 Package Structure

The SEEK prototype code is placed in the *seek* package. Like other module codes in the SEEK prototype, the SA module code resides in the *seek* package. The code for the SA, SE, and the analysis module (AM) reside in *seek.sa*, *seek.se*, and *seek.am* respectively. This package structure helps us to separate the functionality of each module from one another and to encapsulate similar functionality within the same package.

Java classes in the SA package are further divided into sub-packages according to their functionality. These packages are: *seekstructures, seekvisitors, syntaxtree, visitor,* and *parsers* and their content are described below:

The package *seekstructures* contains files that were written for analyzing the Java code. The package *seekvisitors* contains visitor classes that were written to traverse the abstract syntax tree of Java code and SQL statements. The package "*syntaxtree"* contains classes that were created by JTB, and each class in this package corresponds to a non-terminal in the production rules of Java and SQL grammars. The package "*visitor"* contains JTB created base visitor classes. The package "*parsers"* contains Java and SQL parser files. These package structures are shown in Figures 4.2 and 4.3. The classes inside the packages *syntaxtree, visitor,* and *parsers* are automatically created by JTB and JavaCC tools and augmented to fit into seek package structure. The classes inside the packages *seekstructures and seekvisitors* are manually written classes to fulfill the goals of SA module. The *syntaxtree, visitor,* and *parsers* packages have 141, eight and 13 classes respectively. The classes inside these packages will remain the same as long as

the Java and SQL grammars do not change. The *seekstructures* and *seekvisitors* packages have 25 and nine classes respectively and are subject to change as we add new functionality to SA module.

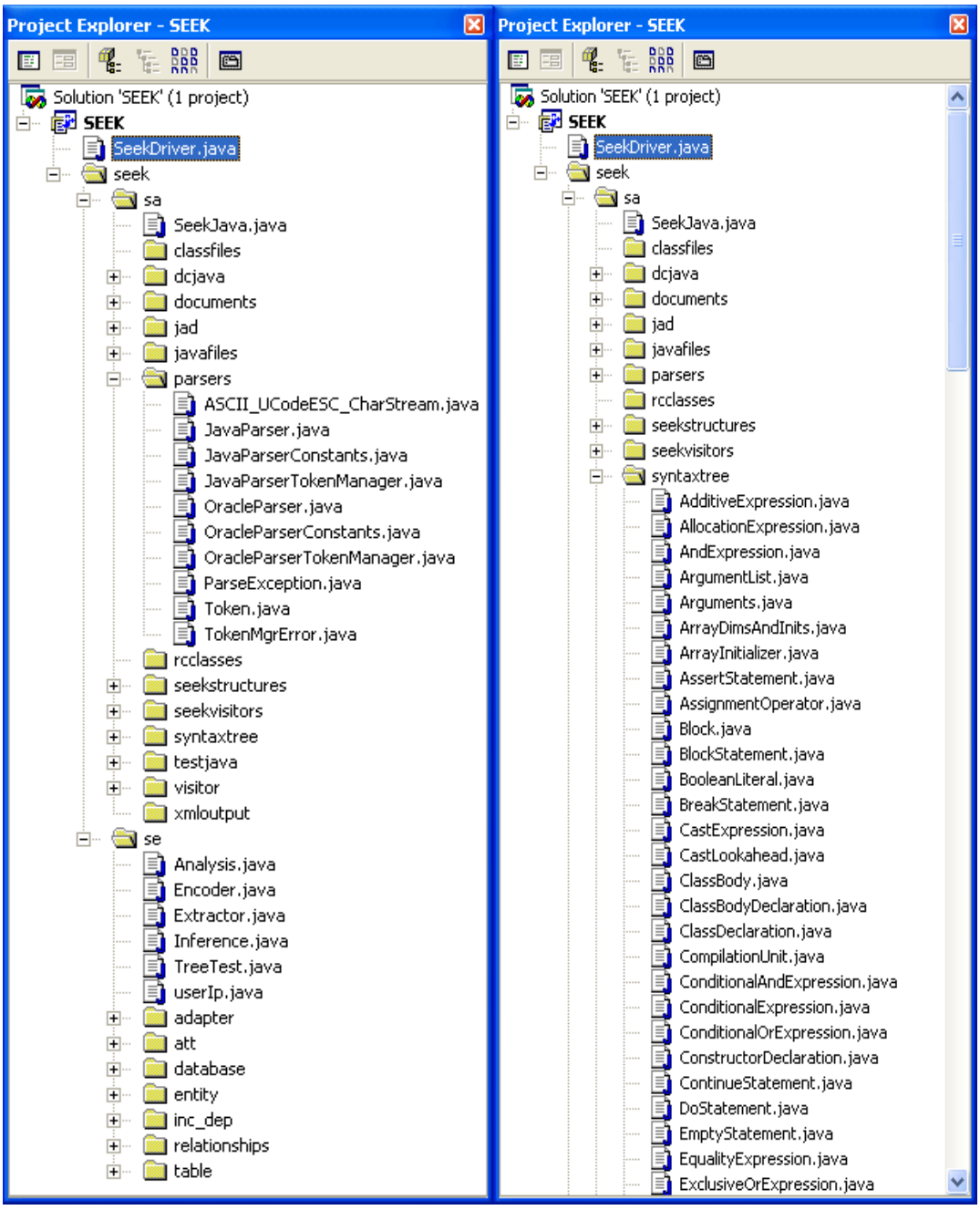


Figure 4-2 Java Semantic Analyzer Packages Structure 1.

The entire package structure is depicted in Figures 4-2 and 4-3. In Figure 4-2, some of the automatically generated classes of *parsers* and *syntaxtree* packages can be seen. Figure 4-3 shows some of the classes of *seekvisitors* and *seekstructures* that we have written.

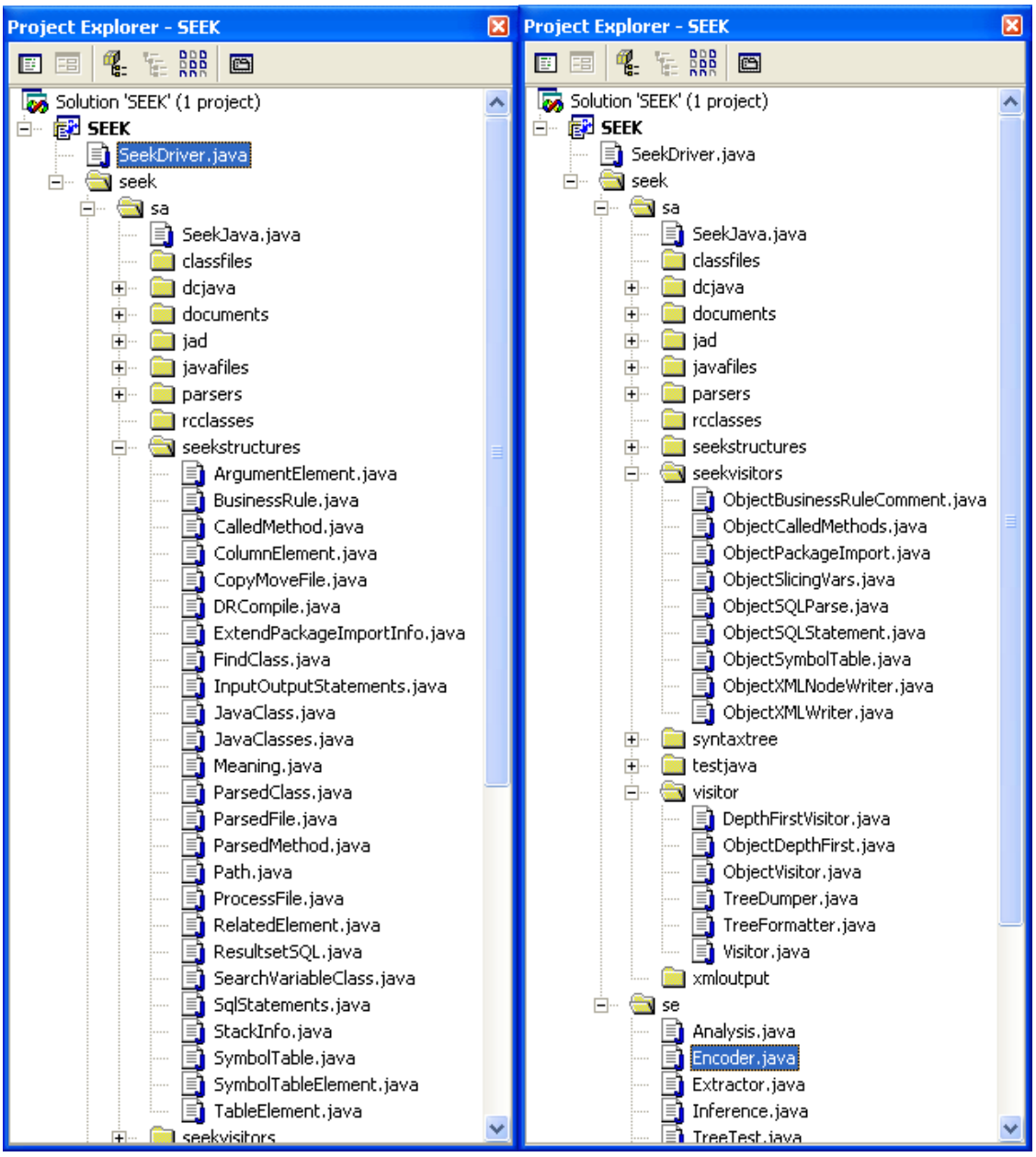


Figure 4-3 Java Semantic Analyzer Packages Structure 2.

As it can be noticed in Figure 4-2 and Figure 4-3 there are other packages in the Java SA package, namely, *classfiles, javafiles, dcjava, rcclasses,* and *xmloutput.* They are

the directories in which we keep the inputs and outputs of SA. The directory "*classfiles"* stores .class files that are being analyzed. The directory "*javafiles"* stores .java files that are being analyzed. The directory "*rcclasses"* stores recompiled classes. The directory "*dcjava"* stores .java files that were generated by decompiling classes by JAD decompiler. The directory "*xmloutput"* stores output files of Java semantic analyzer which contains abstract syntax tree of parsed classes and SQL statements.

4.2 Using the Java Tree Builder To Augment the Grammar Specification File

Implementation started with the augmentation of the grammar specification file with the Java Tree Builder (JTB). As explained in the Chapter 2 and 3, to have a more robust parser, the JavaCC parser generator technology is used. To utilize the benefits of visitor patterns, JTB technology is used to add desired Java code inside the grammar specification file. When the grammar specification file is given to the JTB, a new grammar specification file with visitor patterns is produced. Details of JavaCC, JTB, and visitor patterns are described in Chapter 2.

The grammar specification file (*.jj* file) for Java that was obtained from JavaCC repository [6] is given as an input to JTB 1.2.2. The outcomes are the Java files in the *syntaxtree* and *visitor* directories and also the new grammar specification file, *jtb.out.jj*. Figure 4-4 illustrates the input and outputs of the JTB tool. JTB produces the Java classes in the *syntaxtree* directory according to production rules in the grammar. Each Java class also has an "*accept*" method to be used by visitors. Visitor Java classes that are created in the *visitor* directory have a method that corresponds to each of the Java classes inside *syntaxtree* package. Each of the Java classes are modified to put them into appropriate *seek* packages, *syntaxtree* classes into the *seek.sa.syntaxtree,* and visitor classes into seek.sa.visitor packages. This is done by adding *"Package seek.sa.syntaxtree;"* line at the

beginning of the all Java class files in the *syntaxtree* directory and *"Package seek.sa.visitor;"* line at the beginning of all Java class files in the *visitor* directory.

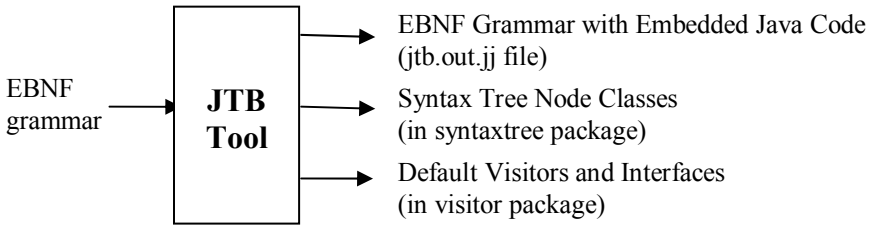


Figure 4-4 Input and outputs of the JTB tool.

4.3 Generate Parser by JavaCC

The output of the JTB tool, the grammar specification file, is input to JavaCC. JavaCC generates the parser files that will accept grammar according to the input specification file. The parser files are listed in Table 4-1.

Table 4-1 JavaCC generated parser files.

ParserName.java	Has main method, methods for each non-terminals, debug(trace) methods, getNetToken method.
ParserNameConstants.java	Interface for tokens
ParserNameTokenManager.java	Manages the token source file
ParseException.java	Exceptions
SimpleCharStream.java	ParserNameTokenManager uses
Token.java	Describes the input token stream
TokenMgrError.java	Returns a detailed message for the Error when it is thrown by the token manager to indicate a lexical error.

For the SA component of the SEEK prototype, two parsers are created, namely a Java and an SQL parsers. Both parsers use the same *SimpleCharStream*, *Token*, *TokenMsgError*, and *ParseException* classes and they have their own ***ParserName,*** ***ParserName****Constants*, and ***ParserName****TokenManager* classes. These parser files are placed in the *parsers* directory for a better package organization. To put them in the

proper package structure, the "*Package seek.sa.parsers*" line is placed at the beginning of each file.

4.4 Execution Steps of Java Semantic Analyzer

The driver class for the semantic analyzer is the *SADriver* class. The *SADriver* accepts the name of the source code file as an argument. The analyzing process starts with this source code file. The *SADriver* invokes the recursive *processFile* method of the *ProcessFile* class with its initial parameters. The parameters of the *processFile* method are *class name, method name*, and *arguments of method*. Initially, parameters of the *processFile* are *source file name* as class name, *main* as a method name, and *String[]* as the argument of the *main* method. It is assumed that the class name parameter that is given to the *SADriver* is the name of the stand-alone Java class, which has the main method. Starting from this stand-alone file, all user-defined classes on the control flow graph are traversed during the analyzing process.

As it is illustrated in Figure 4-5, the SADriver.java takes *Class Name* as a parameter and invokes the *ProcessFile* method on *ProcessFile* class with the parameters *Class Name*, *Method Name* (main), and *Arguments* (String[]).The *ProcessFile* method executes two main steps:

- **Step 1:** The *ProcessFile* method makes preparations necessary for analyzing the source code. Starting from *Class Name* and *main* method of that class, *ProcessFile.java* finds the files that are going to be analyzed with the help of the *FindClass.java*. Then, it copies them into proper directories and extracts the control flow graph by traversing the source code classes.
- **Step 2:** The *ProcessFile.java* starts analyzing the methods in the control flow graph. These two steps are shown in bold face in the *ProcessFile.java* box of Figure 4-5 and are described in the following sections.

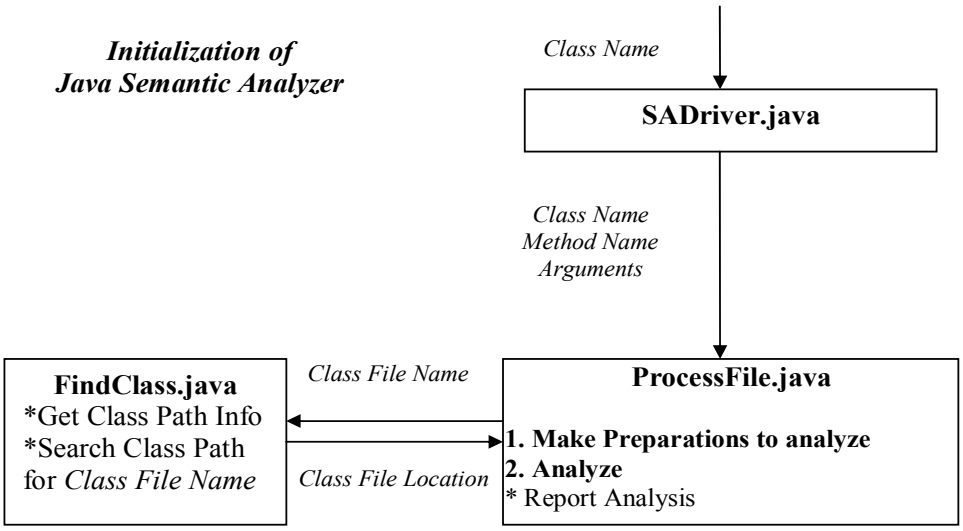


Figure 4-5 Initialization of the Java Semantic Analyzer.

4.4.1 Making Preparations to Analyze

There are two main steps in the process of analyzing the Java code. Details of an iteration of the first step "*Make Preparations for Analyzing*" is explained in this section.

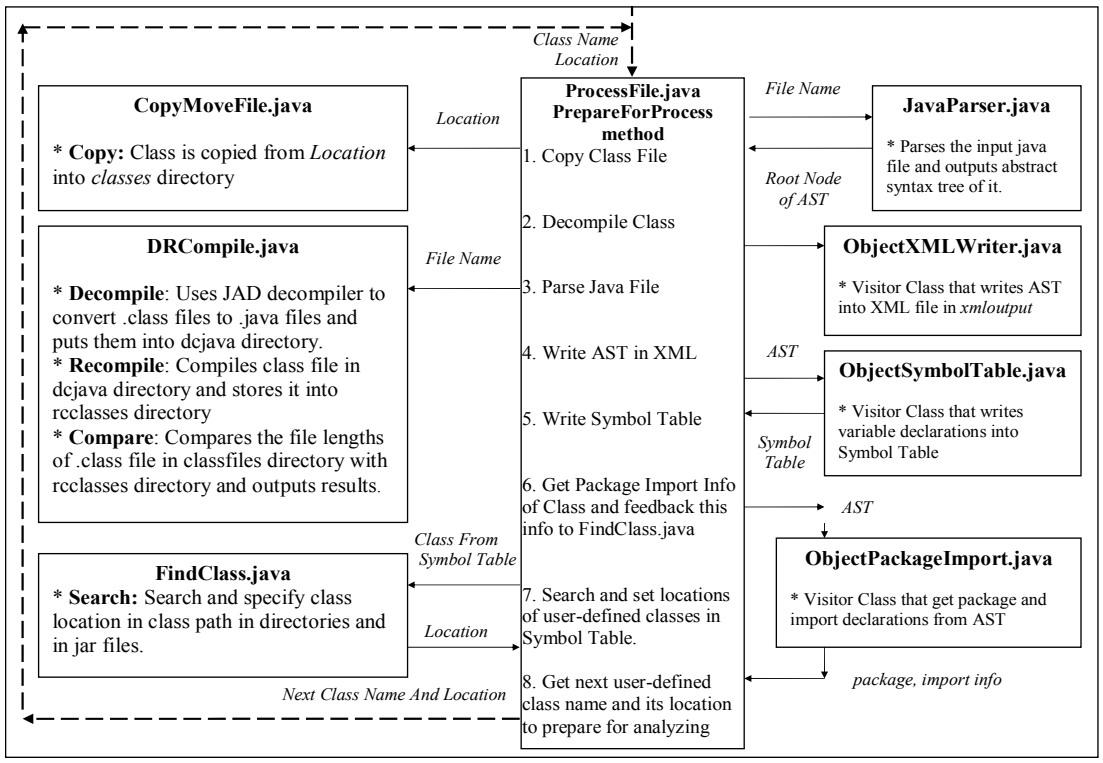


Figure 4-6 Eight steps that make up *prepareForProcess* method.

Figure 4-6 shows the eight steps that make up *prepareForProcess* method of the *ProcessFile* class and its interaction with other classes:

- **Step1:** If the method gets *".class"* as the value of the parameter *extension*, it copies the file from its location into the *classfiles* directory. If the method gets *".java"* as the value of parameter extension, then it copies the file into the *javafiles* directory.
- **Step 2:** If *".class"* is the value of the parameter extension, it decompiles this "*.class*" file by JAD decompiler to get the corresponding source code. This "*.java*" file can then be recompiled by *javac* to compare the sizes of found the "*.class*" file and recompiled the "*.class*" file. The rationale behind this process was explained in the decompiler section of Chapter 3.
- **Step 3:** The ".java" file in the *dcjava* directory or in the *javafiles* directory is parsed by the Java parser.
- **Step 4:** The outcome root node of the Java abstract syntax tree is then passed to the XML Writer visitor class to write the AST in XML format into a file.
- **Step 5:** After that, variable information (name, scope, type, etc.) is extracted from AST by *ObjectSymbolTable* visitor class.
- **Step 6:** Package and import information is also extracted by *ObjectPackageImport* visitor class to be given to the *FindClass* class to be used for next user-defined class finding process.
- **Step 7:** The *SearchVariableClass* class is used to identify the locations of user defined classes in the symbol table. This location information is entered in the symbol table.
- **Step 8:** After all user defined classes are searched in the class path and in the imported packages, it gets the next user defined class name and location from symbol table to prepare it to be analyzed. By these class name and location parameters *prepareForProcess* method is called recursively.

The process of preparing classes continues until all eight steps are applied to all classes in the control flow graph. Visitor patterns are used during the fourth, fifth, and sixth steps. Details of these steps will be given in Section 4.5.

4.4.2 Analyze Source Code

Following initialization, the SA analyzes the methods. Figure 4-7 shows what is being done in an iteration of the recursive *analyzeMethods* method of *ProcessFile* class.

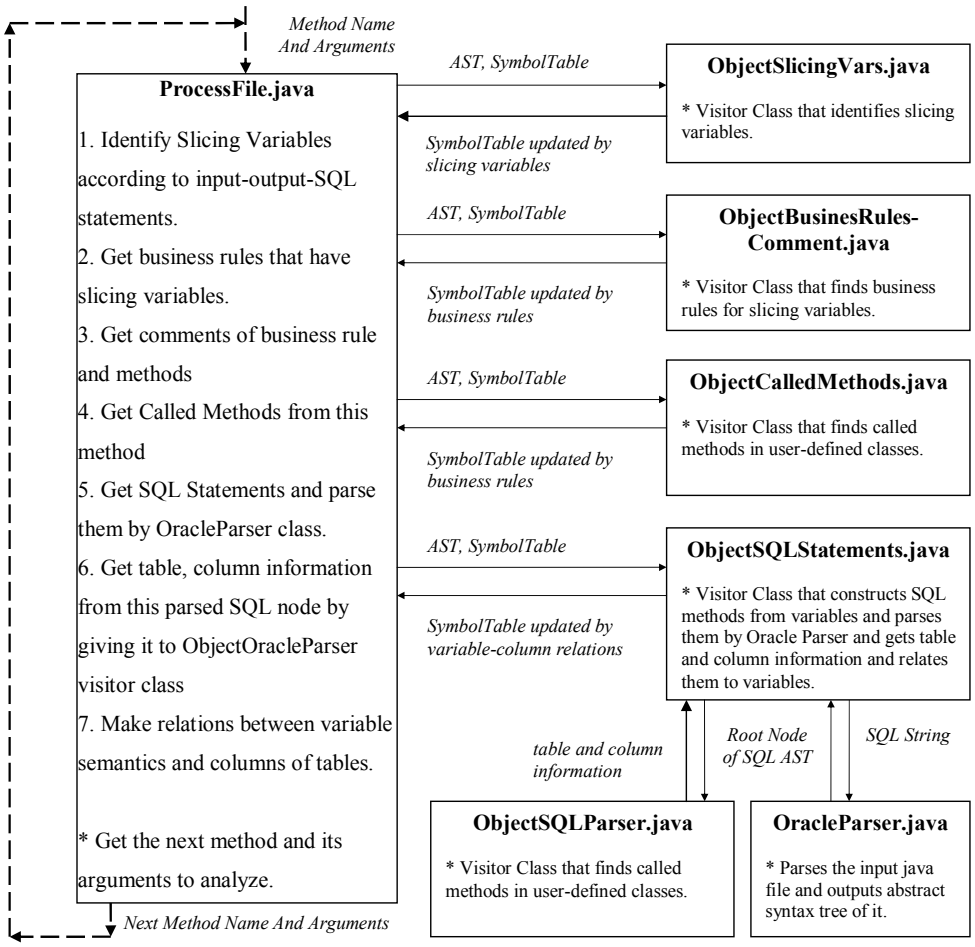


Figure 4-7 Seven steps that make up *analyzeMethods* method.

As it is shown in Figure 4-7, the analyzing source code has seven steps, which are:

- **Step 1:** Identify slicing variables by O*bjectSlicingVars* visitor pattern.
- **Step 2:** Extract the business rules statements that these slicing variables are used in by *ObjectBusinessRuleComme*nt visitor pattern.
- **Step 3:** Extract the comments[1] of the business rules and methods by *ObjectBusinessRuleComment* visitor pattern.
- **Step 4:** Extract method calls on user-defined classes and forms a link list for the called methods by *ObjectCalledMethod* visitor pattern.
- **Step 5:** Constructs the SQL strings by *ObjectSQLStatement* visitor pattern and parses them.
- **Step 6:** The AST of the SQL string is then sent to *ObjectParseSQL* visitor class to get table, column information.
- **Step 7:** This information is used by *ObjectSQLStatement* visitor pattern to make relations between slicing variables and column names.

[1] We can extract comments only if we are processing *.java* files. When we process *.class* files, we do not have access to comments since they can not be reproduced by de-compilation step.

After the seventh step is executed, the *analyzeMethods* method gets the next called method name and arguments, and calls itself with this method name and its arguments recursively. Figure 4-7 also shows the interaction of *analyzeMethods* method with the classes it uses. The details of the visitor patterns that are used in this process are provided in the following section.

Note that, we first process the class files as a whole in the preparation step. Then, we only analyze the methods that are used by method calls. As it was stated in Chapter 3, this approach save effort and improve performance. We do not analyze methods that will never be called.

4.5 Functionality of Visitor Classes

In this section, we define the functionality of the visitor classes. Visitor classes help to extent the functionality of our code easily. If we need a further functionality, all we need to is to write a new visitor class that covers this functionality. Visitor classes also help to organize the related functionality in the same visitor class.

Visitor classes can be written by extending a default visitor class and by overriding methods of this default visitor class. The default visitor class is *ObjectDepthFirst* class, which implements visitor interface *ObjectVisitor*. When a method is not overwritten, it uses the method of the extended class, *ObjectDepthFirst*. Methods of the default visitor only help navigate between methods. Next, we explain the functionality of the visitor classes that is used in our project.

4.5.1 Output Abstract Syntax Tree to XML File

ObjectXMLWriter is the visitor class that is used to output Abstract Syntax Tree (AST) of a Java file into an XML file. Node names of the AST form tag names of the

XML file. All XML files are written into *xmloutput* directory. *ObjectXMLWriter* visitor is used to write the AST of SQL statements as well.

4.5.2 Get Variable and Method Declaration Information

ObjectSymbolTable is the visitor class that is used to gather variable declaration information. It constructs the main file-class-method organization structure that will be used in follow-on steps of the analyzing process. This structure is shown in Figure 4-8.

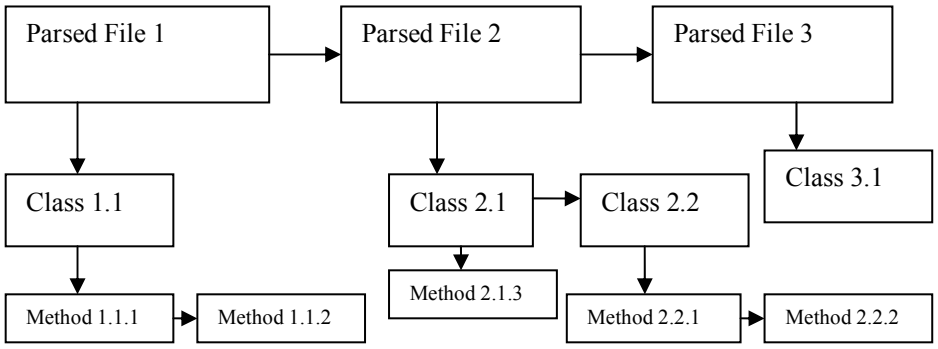


Figure 4-8 Link List Structure of Traversed Method Tree of SA.

In Figure 4-8, *Parsed File1* is the file of the first class that was given as a parameter to SA. In general, Java files can contain more than one class definition. The *Parsed File 2* shown in Figure 4-8, is an example of this type of file. It has classes *Class 2.1* and *Class 2.2*.

The *ObjectSymbolTable* visitor stores variable declarations in a symbol table. Locations of variables in the symbol table are specified with the help of a hash table. Method and constructor declarations are also stored in the symbol table. The formal parameter list of the method and constructors are attached to them with a *nextProcElement* pointer. When a method declaration is inserted into the symbol table, its scope and stack information is also saved. Scope and stack information are important when we analyze two or more different methods of a class that have variables with the

same names. After we gather the meaning of a variable and save it into the symbol table, we must not override that meaning with the meaning of a variable that has the same name but declared in another method. Keeping scope and stack information class-wise helps us to distinguish it from other variables with the same name but in different scopes. Scope is changed by opening a new scope and closing the existing one at the beginning and end of the visitor methods. According to Java grammar, scope is changed inside the visitor method of the following AST nodes:

- Class Declaration,
- Method Declaration,
- Constructor Declaration, and
- Statement.

ObjectSymbolTable visitor stores a *SymbolTableElement* class into symbol table that holds information for each variable and method. The information in this class is then enriched by other visitors. Table 4-2 outlines the information that is stored in *SymbolTableElement*.

SymbolTable and *SymbolTableElement* are the most important class in the SA. *SymbolTable* class is passed to each visitor pattern that aims to extract information from code. The information held in *SymbolTableElement* is enhanced by each visitor pattern.

4.5.3 Get Slicing Information

ObjectSlicingVars is the visitor class that is used to identify slicing variables. These are important variables in the process of semantic extraction. As previously defined in Section 3.6, slicing variables are used in input, output, and SQL statements. They are also passed to a method as a parameter. When we encounter a variable as an argument to an input, output, SQL, or user-defined method, we set the appropriate flag in the

SymbolTableElement structure to indicate that the meaning of this variable should be investigated further.

Table 4-2 Information stored for variable/method declaration in SymbolTableElement.

Info Name	**Definition**
Name	Name of the variable
Scope	Scope of the variable
DeclarationNo	declaration sequence number in the class
InputSlicing	is set true if the variable is used in an input statement
OutputSlicing	is set true if the variable is used in an output statement
SQLSlicing	is set true if the variable is used in an SQL execute statement
MethodSlicing	is set true if the variable is passed as a parameter to a method
ResultsetSlicing	is set true if the variable is used in a resultset get method
UserDefinedClass	is set true if type of the variable is a user defined class
ClassLocation	Location of the user defined class. Class is the type of the variable
Category	If it is variable declaration, value of category is 'variable', otherwise it is 'method' or 'constructor' depending on the declaration.
Type	Class of the variable
Value	Used to store string value of string slicing variables
NextElement	If there is more than one declaration with the same name but different scope, new variable is added to the end of SymbolTableElement list.
NextProcDec	Pointer to the list of formal parameters of the method
RelatedElement	If there is an assignment statement between two variables, they are related with this pointer
ColumnElement	if there is an assignment from resultset get method to this variable, points to the corresponding columnElement in the resultset get method
StackPosition	stores the stack scope info when an element of category method is defined

4.5.4 Get Business Rules and Comments

ObjectBusinessRuleComment is the visitor class that extracts a list business rules and comments related to this business rules. The visitor class searches business rules inside *assignment*, *if,* and *switch* statements. Such a statement is interpreted as a business rule if it contains a slicing variable. The Table 4-3 provides information that is stored in the *BusinessRules* class.

Slicing variables that exist in this business rule are saved in *RelatedElement* field of business rule structure. This visitor also saves the comments in the *BusinessRule* class that occur just before the potential business rules in the code. The pointer to the root of the node of the business rule in the AST is also saved. This node can be used in outputting this business rule in a desired format.

Table 4-3 Information stored in BusinessRule class.

Info Name	**Definition**
Type	Type of the business rule, like Statement, Assignment
Text	Text of business rule
Comment	Comment that were placed right before the business rule text
ClassName	Class name in which business rule is
MethodName	Method Name in which business rule is
RelatedElement	The pointer to the list of variables that exists inside this business rule
Node	The pointer to the root of the node of the business rule in the AST
Next	Next business rule

One business rule can reside inside another. In such a case, the business rule is inherited inside another. If the business rule is inherited by another business rule, we concatenate the current business rule (the one that is inherited) with the previous business rule (the one that inherits) and also add the slicing variables of the current rule to the previous rule. This scenario is depicted in Figure 4-9. *business rule 1* inherits *business rule 2*. *var1,var2, var3, var4, var5,* and *var6* are slicing variables of *business rule 1*, while *var4, var5,* and *var6* are the slicing variables of *business rule 2*.

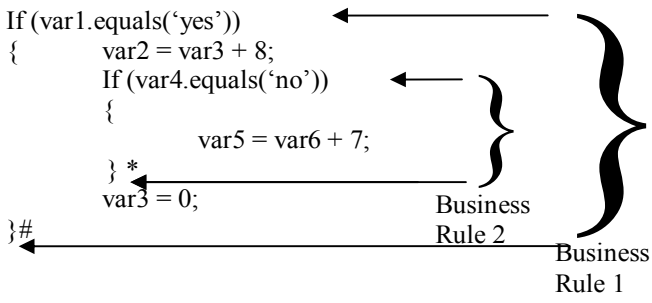


Figure 4-9 Example of a business rule that inherits another business rule.

The elicitation of business rules is achieved using stacks. When we encounter a potential business rule statement (*if, assignment, or switch statement*), we begin to store the tokens of the code in a stack. Meanwhile, if we encounter a slicing variable, we also store the slicing variable in a stack. When a new business rule is encountered before the end of the one currently under investigation, a new business rule is pushed onto the stack. Figure 4-10 illustrates the contents of the stack at the point when "*" in Figure 4-9 is reached. *BR2* in Figure 4-10 contains the *"If (var4.equals('no')) {var5 = var6 + 7;}"* string, and *BR1* in Figure 4-10 contains the "*If (var1.equals('yes')){var2 = var3 + 8;"* string. When the latter business rule has been processed, the text and slicing variables for it are added to the previous one. The content of the stacks at the point when "#" in Figure 4-9 is reached are shown in Figure 4-11. It can be readily seen that, the content of the *VS2* stack is transferred to *VS1* and the text of *BR1* and *BR2* are also merged. The *BRText* contains the "*If (var1.equals('yes')){var2 = var3 + 8; If (var4.equals('no')) {var5 = var6 + 7;} var3 =0;"* string.

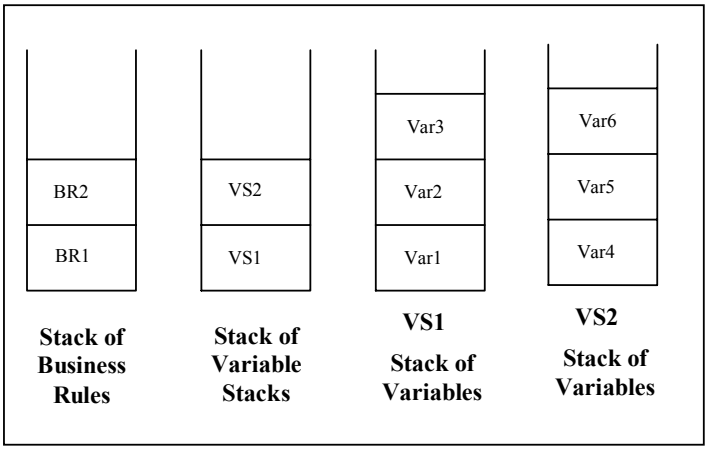


Figure 4-10 The content of the stacks at point when "*" in Figure 4-9 is reached.

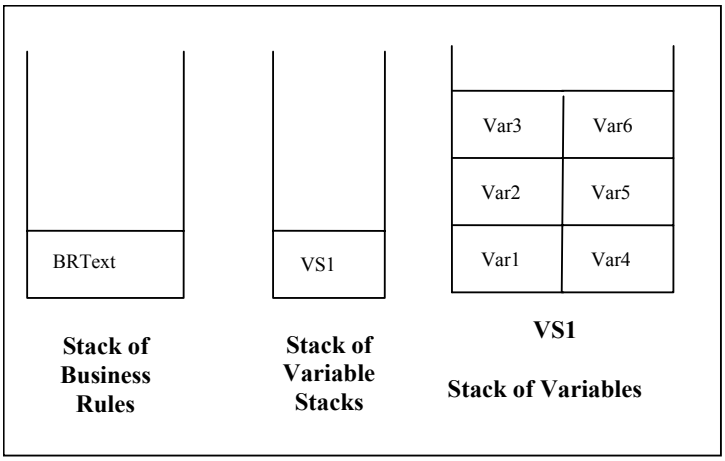


Figure 4-11 The content of the stacks at point when '#' in Figure 4-9 is reached

Another functionality of *ObjectBusinessRuleComment* visitor class is that it saves comments that are written just before the method declarations in the code. These comments are saved and then passed to the method that is being analyzed.

4.5.6 Get Method Calls

The visitor class *ObjectCalledMethods* gets the list of method calls inside a method. These methods are the ones that are invoked on user defined classes. When the visitor *ObjectCalledMethods* encounters a method, it adds the method to the list with its method name, class name, and argument list. The argument list is formed from the actual parameters of the method call. An argument element class that stores the properties of an argument has name, type, and value fields. Figure 4-12 outlines the structure of the method call and argument list that are gathered by *ObjectCalledMethods* visitor.

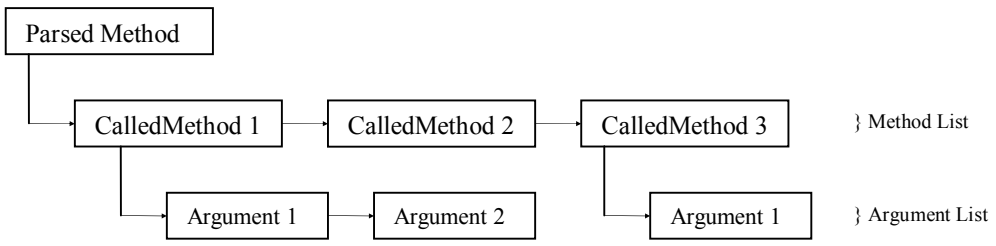


Figure 4-12 List of Method Calls Structure.

4.5.7 Get Package-Import-Extend Info

The *ObjectPackageImport* is the visitor class that is used to obtain package, imported classes, and extended class information of a Java file. This information is then used in finding the user-defined classes in the class path using the file finder class *FindClass*.

4.5.8 Get SQL Statements

The *ObjectSQLStatement* is the visitor class that is used to extract SQL statements and parse them. This is one of the core visitors of the SA. While the visitor traverses the AST, it constructs the value of string variables. When a variable of type string or a string text is passed as a parameter to an SQL execute method (e.g., *executeQuery(queryStr)*), it parses the string with SQL parser. The SQL parser returns the AST of this SQL string. The *ObjectSQLStatement* visitor then uses the *ObjectSQLParse* visitor to get information about that SQL statement. The details of the *ObjectSQLParse* visitor are discussed in the next section. If the value of a variable is compared with a column in the where clause of an SQL statement, it saves information about the relation between the variable and the column. While the visitor traverses AST, if it encounters a SQL retrieve method (*like getString("column name")*) on a *Resultset* type of variable instance, it gets the related column info and relate it to variable name. If *ObjectSQLStatement* encounters an output method, it evaluates the string by relating possible meaning with the variable names. It relates the variables with the text in the most recent output statement.

4.5.9 Parse SQL Strings

The *ObjectSQLParse* is the visitor class that is used to parse SQL strings and store the information about the SQL string into a *ResultsetSQL* class structure. Table 4-4 lists the members of the *ResultsetSQL* class.

Table 4-4 Information stored in ResultsetSQL structure

Info Name	Definition
Name	Class Name + Method Name + Declaration No
Type	Type can be 'update', 'select', 'insert' or 'delete'
String	SQL string
Node	Root node of the AST of the SQL string
Columns	Pointer to column list
Tables	Pointer to table list

The AST of an SQL select statement is given to the *ObjectSQLParse* visitor with an initially empty *ResultsetSQL* class instance. The *ObjectSQLParse* visitor traverses the AST of the SQL statement and extracts table, and column names. In addition, the *ObjectSQLParse* fills the *ResultsetSQL* class with the column and table information. As one can see in Figure 4-13, when a SQL statement is forwarded to *ObjectSQLParse* visitor, a list of column and table information is attached to the *ResultsetSQL* instance. The *ObjectSQLParse* can extract table and column information of *Uptade, Insert,* and *Delete* statements as well.

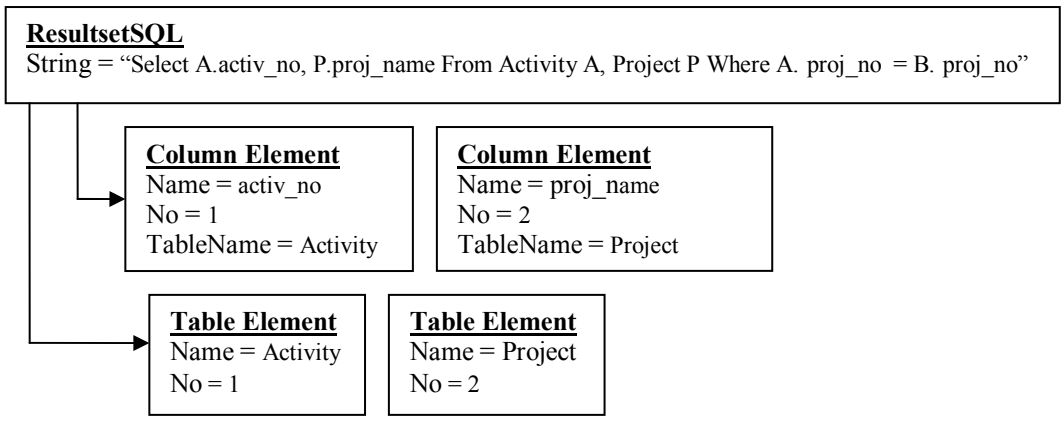


Figure 4-13 Example of a ResultsetSQL class.

CHAPTER 5
EVALUATION OF SEMANTIC ANALYZER

Several criteria can be used to evaluate our semantic analyzer. The main criteria include correctness and performance. In this chapter, we introduce our prototype test system to demonstrate how SA captures the semantics from source code correctly and within a reasonable amount of time. In Section 5.1, we show the correctness of key features of SA by using code fragments. In Section 5.2, we show the semantics extracted from an application from *Construction Scheduling* and discuss the correctness of these semantics. In Section 5.3, we show the performance improvement in time when compared to previous version of SA and explain how we achieved this.

5.1 Can Features of Semantic Analyzer Extract Semantics Correctly?

In this section, we use code fragments to demonstrate how key features of SA can correctly extract meaning. In each subsection, we introduce a code fragment and show what SA can extract from that code fragment. Then, we explain why this extracted meaning is correct.

5.1.1 Targeting a Specific Column from the Query

In this subsection, we show that SA can target a specific column from the result set of a query. The correctness of this feature is important to satisfy the eighth heuristic that is introduced in Section 3.7.

When a SQL select query string is given as a parameter to *executeQuery* method of a *Statement* object instance, the tuples satisfying the selection criteria of the query are returned in a *ResultSet* object. The Java Database Connectivity (JDBC) Application

Program Interface (API) provides several methods for ResultSet object to obtain individual column values from a tuple in the ResultSet. The parameter of a ResultSet get method can be one of the following:

1. A string: Column name from the SELECT query column list
2. An integer: Between zero and the number of columns in the query minus one.

SA can accurately handle these two types of parameters of the *ResultSet get* methods and can specify the corresponding column name in the column list of the SQL query statement. SA stores the *ResultSet* information inside an instance of *ResultSetSQL* class. Examples of these two parameters are given in Table 5-1.

Table 5-1 Examples of ResultSet get method parameters.

ResultSet get method parameter as an integer: ResultSet resultSet = sqlStatement.executeQuery("*SELECT p_id FROM Project*"); String projNumber = resultSet.getString(1);
ResultSet get method parameter as a String: resultSet = sqlStatement.executeQuery("*SELECT p_id FROM Project*"); String projNumber = resultSet.getString('p_id');

As shown in Figure 5-1, the column list and table list are extracted from the SQL query string and a list of *ColumnElement* and *TableElement* are formed and attached to the *ResultSetSQL* class instance. *ResultSetSQL* is formed by *ObjectSQLParse* visitor and is introduced in Section 4.5.9. When the "get method*"* parameter is an *integer*, we find the *(integer parameter +1)*th element in the *ColumnElement* list. When the "get method" parameter is a *String*, we find the column by comparing the name of the parameter with the name of the elements in the *ColumnElement* list.

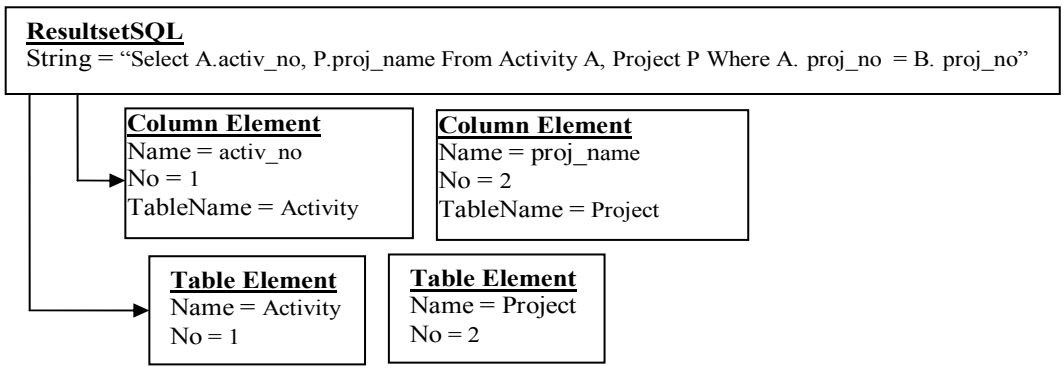


Figure 5-1 Example of a *ResultsetSQL* class and *ColumnElement* and *TableElement* lists.

After representing the code examples in Table 5-1 in a ResultSetSQL class form as shown in Figure 5-1, SA can correctly target the specific column as explained in the previous paragraph. Correctly targeting a specific column feature is important to relate the correct column of a table with the correct output statement which will give the meaning of the column.

5.1.2 Extracting the Column Name from Mathematical Expression

In this subsection, we show that SA can extract the name of the column even if it is embedded in a complex mathematical expression. The correctness of this feature is also important to satisfy the eighth heuristic that is introduced in Section 3.7.

Table 5-2 Example of a SQL query that has mathematical expression in its column list.

```
resultSet = sqlStatement.executeQuery("SELECT (activ_planned_finishdate+1)
                FROM Activity WHERE activ_no = 5");
java.sql.Date startDateSQL = resultSet.getDate(1);
```

Figure 5-2 gives the *ResultSetSQL* class representation of the query in Table 5-2. As shown in Figure 5-2, *ColumnElement*'s *name* field has the correct column name. As it was stated before, *ResultSetSQL* is formed by *ObjectSQLParse* visitor. *ObjectSQLParse* traverses the AST of SQL expression and accurately extracts the column name.

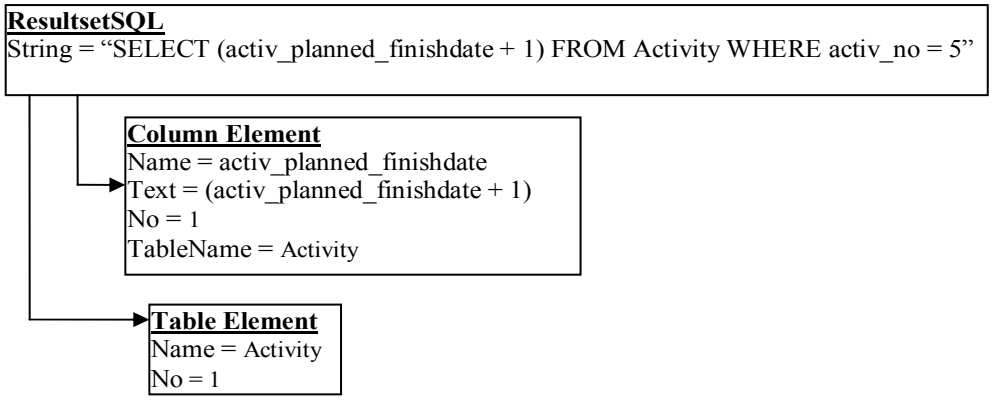


Figure 5-2 ResultSetSQL class representation of the query in Table5.2.

Even we find a meaning related with the mathematical expression, if SA could not identify the column name from a mathematical expression, then SA would fail to relate meaning with the correct column name. If SA could not identify the column name from a mathematical expression, SA would not find a matching entry to fill with the meaning in the schema of data source that was extracted by SE.

5.1.3 Geting Query Expression from the Variable Value

In this subsection, we show that SA can correctly save and retrieve the value of a *string* variable. The parameter given to the "SQL execute method" (e.g., *executeQuery*) can be a *string* type of variable as well. The *string* variable can hold an SQL query. The value of the variable must be correctly saved and retrieved when it is used in an SQL execute method.

Table 5-3 Example of using variable in the *executeQuery* method.

```
String queryString = "SELECT p_id FROM Project";
ResultSet resultSet = sqlStatement.executeQuery(queryString);
String projNumber = resultSet.getString(1);
```

An example of using a variable as a parameter of "SQL execute method" is given in Table 5-3. Note that, there can be several numbers of statements between the value assignment of variable and the SQL execute method.

5.1.4 Concatenating Strings to Get the Query Expression

In this subsection, we show that SA can correctly recognize the meaning of concatenating operator (+) and concatenate strings to get the query expression. Table 5-4 is an example of this. When *query1* and *query2* in Table 5-4 are declared, their initial values are stored in the *value* field of their *SymbolTableElement* in the *SymbolTable*. When they are used in a string expression with the concatenating operator, their values are obtained from *SymbolTable* and are used to form the new string.

Table 5-4 Example of Concatenating String to Get the Query Expression.

```
String query1 = "SELECT active_duration, activ_planned_startdate " ;
String query2 = " FROM Activity " + " WHERE Activ_no = '5' ";
String query = query1 + query2 +
ResultSet resultSet = sqlStatement.executeQuery(query);
```

Figure 5-3 illustrates how variable values in Table 5-4 are stored in *SymbolTable*. The value of the element *query2* is formed by concatenating the strings *"FROM Activity",* and *"WHERE Activ_no = '5' "*. The value of the element *query* is formed by concatenating the value of string variables *query1* and *query2*.

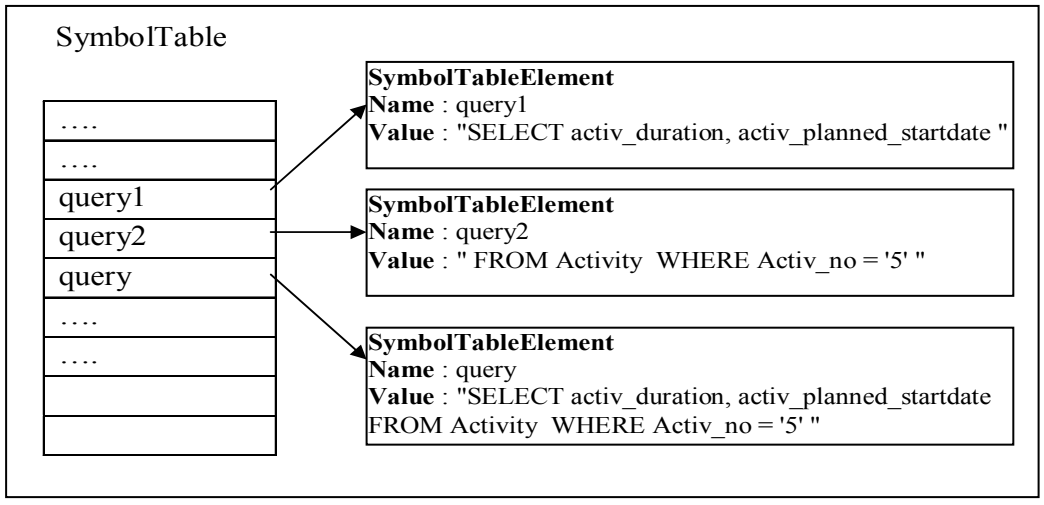


Figure 5-3 Illustration of String *SymbolTableElement*s and their values.

SA can correctly form the *string* expression by concatenating any number of *string* variables and expressions. These variables and expressions can form an SQL expression or an input/output statement that inherits meaning. Correctly forming and evaluating the SQL expression and input/output statement is important to get the most benefit from the application code.

5.1.5 Geting the Meaning of Slicing Variable from the Output Text

SA can accurately relate output statements with slicing variables. The slicing variable concept is introduced in Section 3.6. SA gets the meaning of the slicing variables from the most recent output text. To achieve this, SA pushes the output text in a stack and when a slicing variable is encountered, it pops the text from the stack and relates it with that slicing variable.

Table 5-4 Making relations between output statements and columns.

```
String query = "SELECT activ_duration,active_planned_startdate FROM Activity
                                    WHERE activ_no = '"+ inputActivity + "'";
resultSet = sqlStatement.executeQuery(query);
duration = resultSet.getInt(1);
oldStartDate = resultSet.getString(2);
System.out.println("activity no: " + inputActivity);
System.out.println("old start date: "  + oldStartDate);
System.out.println("duration: " + duration);
```

In the example in Table 5-4, the meaning of the *inputActivity, oldStartdate,* and *duration* slicing variables are obtained from the most recent output text and these meanings are then related to *activ_no, activ_duration,* and *active_planned_startdate* variables. Note that *inputActivity* variable is used in the *where* clause of the query. SA also keeps track of the variables that are used in *where* clauses because they have potentially important information about the semantic of their correspondence columns in the *where* clause.

Correctly finding the meaning of the slicing variable is important because after finding the meaning of the slicing variable, we assign this meaning to the column of a table that was already related with that variable. Correctly realizing this feature means that we can correctly realize the eighth heuristic in Section 3.7.

5.1.6 Passing Semantics from one Method to Another

One of the most powerful features of SA is passing argument values and meanings from one method to another. SA conquers the meaning found in the current method with the meanings found before the current method was called. SA transfers semantics from one method to another by this feature causing it to get more accurate results. When a method is called, the values of parameters are passed from the point where the call is made to the called method. The execution continues from the called method.

In the example given in Table 5-5, the SQL query information is passed to the *outputProjectResultset* method in the *resultSet* variable. Although the SQL query is in the *initializeQuery* method and the output statement is in the *outputProjectResultset* method, SA can accurately keep track of the parameters that are passed between methods and can relate *"Project Num:"* output statement with *p_id* column of table *Project*. If SA could not correctly transfer the semantics from one method to another through arguments, then SA would not be able to capture the semantics of p_id column of table *Project* and this semantics would have been lost.

5.1.7 Extracting the Meaning from the Title Output

Slicing variables may not have output statements that identify their meaning. Instead, they may have a title output statement as shown in Table 5-6. In this case, SA shows the title to the user as a possible meaning. The title inherits semantics for the following outputs of columns. In the example given in Table 5-6, *projectNo,*

workItemNo, and *currentactivity* variables and their corresponding columns have *"proj_no workitemno activityno"* output statement as a possible meaning. Note that, texts like *"----------"* and *"\t"* are used for formatting. They are eliminated and are not shown as a possible meaning. Correctly identifying the title of the outputs of columns and presenting this title to user helps us to get more semantic information from the application code.

Table 5-5 An example of passing parameter values.

```
Public static void initializeQuery (Statement sqlStatement,ResultSet resultSet)
{
  String query = "SELECT p_id FROM Project";
  resultSet = sqlStatement.executeQuery(query);
  outputProjectResultset(sqlStatement,resultSet);
}

public static void outputProjectResultset (Statement sqlStatement,ResultSet resultSet)
{
System.out.println("The following are all the projects found:");
while(resultSet.next())
{
        String projNumber = resultSet.getString(1);
        System.out.println("Project Num:" + projNumber);
}
resultSet.close();
}
```

Table 5-6 Extracting the meaning from the title.

```
query = “SELECT Activ_no, Workitem_no, Proj_no FROM Activity";
resultSet = sqlStatement.executeQuery(query);
System.out.println("proj_no workitemno activityno ");
System.out.println("-------------------------------------");
while(resultSet.next()){
        projectNo = resultSet.getString(3);
        System.out.println(projectNo);
        workItemNo = resultSet.getString(2);
        System.out.print("\t" + workItemNo);
        currentactivity = resultSet.getString(1);
        System.out.print("\t" + currentactivity);
}
```

The correctness of the above key features of SA is important to accomplish the goal of SA. SA correctly extracts meaning from code as introduced in the above examples and extracts the highest amount of semantics that is available in the source code.

5.2 Correctness Evaluation

In this section, we show how SA accurately extracts the application specific meaning from source code. To show this, we use an application from *Construction Scheduling*. This program does a very simple re-scheduling of an activity of a construction firm. The source code can be found in Appendix G. Table 5-7 shows the list of meanings of the columns that are extracted from the *Construction Scheduling Program* source code. Now, we explain how SA correctly satisfies our heuristics and extracts the meanings listed in Table 5-7.

The reader can notice that some of the columns have only one meaning and some of the columns have several meanings in Table 5-7. If all the output texts are the same or there is only one output text for a column of a table, then there is only one meaning assigned to that column. In this case, the column meaning has no ambiguity. In Table 5-7, the meaning of *p_id* column of table *Project* and the meaning of *active_duration* column of table *Activity* are found without any ambiguity. If more than one output statements can be associated with a column, then there is ambiguity. *Activ_no, activ_planned_finishdate*, and *activ_planned_startdate columns* of table *Activity* have several meanings assigned to them. For this reason, there is ambiguity in the meaning of these columns.

When there is ambiguity, the list of meanings is presented to the user and an input to resolve the ambiguity is expected. The frequency of some words can give clues about the meaning of the column. For instance, the word *activity* has the highest frequency in the list of meanings for *activity_no* column of *Activity* table. Even if the column name

was irrelevant, instead of *Activity_no*, the frequency of word *activity* could be a good clue to identify its meaning.

Table 5-7 Extracted Meanings of the Columns from the Construction Scheduling Program Source Code.

Column & Table Name:	**Extracted Meanings:**
Column Name: Activ_no Table Name: Activity Ambiguity: true	1) projno workitemno activityno startdate finishdate 2) Activity is being rescheduled to the earliest date possible 3) activity no 4) You have entered days to shift the activity 5) Activity 6) Sorry there is no slot to shift the activity 7) You can not choose the same activity 8) Sorry there is no enough slot to place the activity 9) Sorry there is no enough slot to place the activity after the activity
Column Name: p_id Table Name: Project Ambiguity: false	1) Project Num
Column Name: Proj_no Table Name: Activity Ambiguity: true	1) projno workitemno activityno startdate finishdate
Column Name: Workitem_no Table Name: Activity Ambiguity: true	2) projno workitemno activityno startdate finishdate
Column Name: activ_planned_finishdate Table Name: Activity Ambiguity: true	1) projno workitemno activityno startdate finishdate 2) The new finish date is 3) The new start date is 4) The new start date is
Column Name: activ_planned_startdate Table Name: Activity Ambiguity: true	1) projno workitemno activityno startdate finishdate 2) old start date 3) You can shift this activity at most 4) oldstartdate
Column Name: activ_duration Table Name: Activity Ambiguity: false	1) duration

Another form of ambiguity can be seen when a meaning statement is used for more than one column. For instance, *Project_no and WorkItem_No* columns of table *Activity* have this kind of ambiguity. The code fragment that causes this kind of ambiguity is

shown in Table 5-8. In the code fragment in Table 5-8, a title is given for the preceding outputs. SA eliminates the format outputs such as *"---------"* and *"\t"* and assigns the title meaning *"proj_no workitemno activityno startdate finishdate"* to *Activ_no, activ_planned_startdate, activ_planned_finishdate, Workitem_no,* and *Proj_no* of table *Activity.*

Table 5-8 Extracting the meaning from title of the outputs.

```
query = "SELECT Activ_no, activ_planned_startdate, activ_planned_finishdate,
        Workitem_no, Proj_no FROM Activity ORDER BY activ_planned_startdate";
resultSet = sqlStatement.executeQuery(query);
System.out.println("proj_no workitemno activityno startdate finishdate ");
System.out.println("-------------------------------------------------");
while(resultSet.next())
{
        currentactivity = resultSet.getString(1);
        workItemNo = resultSet.getString(4);
        projectNo = resultSet.getString(5);
        System.out.print(projectNo);
        System.out.print("\t" + workItemNo);
        System.out.print("\t" + currentactivity);
}
resultSet.close();
```

The *Construction Scheduling Program* is also a good example of how SA extracts semantics from applications that have functionality separated into multiple files. The functionality of the scheduling program is separated into three files: *ScheduleMenu.java, ScheduleList.java* and *ManipulateActivity.java.* The analyzing process starts from the *main* method of *ScheduleMenu.java* file and dispatches to the methods in *ScheduleList.java,* and *ManipulateActivity.java* files as it encounters a method call on these classes. Table 5-9 shows a method call that is made on an instance of *ScheduleList* class from the *main* method of *ScheduleMenu.java* class. After SA completes analyzing

initiliazeProjectQuery method, it continues to analyze the *main* method of *ScheduleMenu* from the next statement, *if (menuOption.equals("2"))* statement.

Table 5-9 Code fragment inside the main method of ScheduleMenu.java

```
If (menuOption.equals("1")){
scheduleList.initiliazeProjectQuery(sqlStatement,resultSet); }
if (menuOption.equals("2")){
scheduleList.listActivities(sqlStatement,resultSet);}
```

The *Construction Scheduling Program* source code also verifies that SA can extract and integrate semantic from different methods. SA can transfer the semantic of a variable to a new method when it is used as an argument inside the method call. An example of this case is given in Table 5-10. In this example, the SQL query semantic is held inside the variable *resultSet*. When *listProjects(sqlStatement,resultSet)* method call is done inside the *initiliazeProjectQuery* method, the semantic gathered for *resultSet* variable is transferred to the variable *resultSetProject* in the *listProjects method.* If SA does not have this feature, it can not relate the output statement *"Project Num:"* to the *p_id* column of *Project* table.

Table 5-10 Code fragment from ScheduleList.java shows how variable semantic is transferred from one method to another.

```
Public static void initiliazeProjectQuery(Statement sqlStatement,ResultSet resultSet){
        String query = "SELECT p_id FROM Project";
        resultSet = sqlStatement.executeQuery(query);
        listProjects(sqlStatement,resultSet);}
public static void listProjects(Statement sqlStatement,ResultSet resultSetProject)   {
        System.out.println("The following are all the projects found:");
        while(resultSetProject.next()){
                String projNumber = resultSetProject.getString(1);
                System.out.println("Project Num:" + projNumber);
        }
        resultSetProject.close();
}
```

According to our experiments with *Construction Scheduling Program* source code, SA can correctly satisfy the heuristics that was introduced in Section 3.7 and can correctly extract all the meanings in the user interaction statements that can be related with a column of a table. Analyzing *Construction Scheduling Program* source code has been a good example that validates all the features of the new SA. We have also tested SA with numerous of small code fragments to evaluate the correctness of each of the SA feature. It would be beneficial to test the new SA with several more number of large code samples that can be used to validate all the futures of the new SA in one application.

5.3 Performance Evaluation

SA is a built-time component of SEEK. Although performance is not as critical as correctness, it is important to state that the performance of our new version has significantly improved when compared to the first version [2]. This is because the current version of SA introduces significantly different algorithms to extract information from source code. The current and previous versions of SA are tested on an Intel Pentium-IV PC with a 1.4 GHz processor, 256 MB of main memory, and 512KB cache memory running Windows NT.

Previous version of SA executes four main steps which were explained in [2]. Step 3 and 4 are applied for each of the slicing variables. Table 5-11 gives time spent in seconds for each of these steps and for each slicing variables in step three four while analyzing *Construction Scheduling* program.

- **Step 1:** AST Generator: Constructs the AST for the source code
- **Step 2:** Pre-slicer: Identifies slicing variables
- **Step 3:** Code-slicer: Performs slicing and constructs a reduced AST
- **Step 4:** Analyzer: Traverses reduced AST to extract semantic information

Table 5-11 Time spent (in seconds) in sub steps of previous version of SA.

Step No			**Time Spent in Seconds**
Step 1			1
Step 2			472
Step 3 & Step 4 for each slicing variables	**Slicing Variable Name**	**Time in seconds**	561
	projNumber	4	
	intProjNo	4	
	workItemNo	4	
	currentActivityNo	4	
	start_Date	19	
	finish_Date	14	
	totalActivityNumber	5	
	inputActivity	95	
	oldStartDate	6	
	duration	60	
	inputactivity	14	
	dateDiff	22	
	shiftAmount	12	
	input_Activity	101	
	current_Activity	94	
	old_Start_Date	6	
	dur	48	
	activityCount	3	
	dateDifference	46	
Total Time			1034

For each of the slicing variables, previous version of SA prepares and analyzes a new AST, step 3 & 4 in Table 5-11. It spends significant amount of time to construct the AST for that slicing variable and then extracts information from that AST. If the number of statements that the slicing variable participates is low (e.g., *projNumber*), then the time spent to analyze that slicing variable is small. If the number of statements that the slicing variable participates is high (e.g., *input_Activity*), then the time spent to analyze that slicing variable is large. As the number of slicing variables in source code increases, the total time spent for step 3 & 4 increases.

The new SA executes two main steps which were explained in Section 4.4.

- **Step 1:** Prepare classes.

- **Step 2:** Analyze the methods starting from main method of the given class.

Step 1 prepares all the user-defined classes that are used in the application. Then, Step 2 analyzes the methods. Table 5-12 shows time spent to prepare each of the classes in *Construction Scheduling* program, namely *ScheduleMenu*, *ScheduleList,* and *ManipulateActivity* classes. Note that, time is given in milliseconds.

Table 5-12 Time spent (in milliseconds) in current version of SA.

Step No	**Class or Method Name that is being processed**	**Time Spent**
Step 1 Prepare Classes	ScheduleMenu Class	8406
	ScheduleList Class	2157
	ManipulateActivity Class	10765
Step 2 Analyze Methods	*main* Method of *ScheduleMenu* Class	219
	doDatabaseInitializations Method of *ScheduleMenu* Class	0
	getTotalActivityNumber Method of *ScheduleMenu* Class	813
	menu Method of *ScheduleMenu* Class	31
	initiliazeProjectQuery Method of *ScheduleList* Class	62
	listProjects Method of *ScheduleList* Class	47
	listActivities Method of *ScheduleList* Class	219
	rescheduleActivity Method of *ManipulateActivity* Class	1516
	shiftActivity Method of *ManipulateActivity* Class	187
	changeActivity Method of *ManipulateActivity* Class	1063
Total		29642

The reader can notice that the current version applies Step 1 to each class and applies Step 2 to each method. In the current version of SA, the increase in the number of slicing variables slightly affects the performance, because the current SA does not construct and traverse the AST for each slicing variable. Instead, it constructs the AST for each class and traverses it for each method. While traversing the AST, the current SA considers all slicing variables and analyzes them simultaneously. The total time spent values in Table 5-11 and 5-12 show that we improved the performance of SA by a factor of 35 (1034 / 29.6).

CHAPTER 6
CONCLUSION AND OPPORTUNITIES FOR FUTURE RESEARCH

The SEEK project (Scalable Extraction of Enterprise Knowledge) currently under way in the Database Research and Development Center at the University of Florida has resulted in novel and comprehensive methodologies for semantic analysis of application code to assist in generating a detailed description of a legacy information system. This detailed description of the legacy system is used to overcome problems of assembling knowledge resident in numerous legacy information systems in the business network. The SEEK project aims to enable extended enterprise collaboration in complex business networks. Lack of techniques for scalable extraction of knowledge resident in participating firms in the extended enterprise is an important barrier to such collaboration. In the SEEK project, we develop methodologies and algorithms to discover and extract the detailed description of the legacy system in a flexible and scalable manner that significantly reduces human involvement. SEEK is not meant as a replacement for wrapper or mediator development toolkits. Rather, it complements existing tools by providing input about the contents and structure of the legacy source that has so far been supplied manually by domain experts [1].

The theories and methodologies proposed in this thesis provide a significant improvement to the existing semantic analyzer in SEEK. The previous version represented a comprehensive solution for mining semantic knowledge from application code [2]. This thesis adds heuristics which represent a more extensible and robust approach for extracting semantic information from legacy source. At the same time, our

approach performs significantly better in terms of execution time. Furthermore, this thesis provides a generic solution for the semantic analysis problem for application code written for relational databases. Our approach uses state-of-the-art techniques to obtain the source code when it is only available in the form of object files, to get the AST for the source code, and to extract semantics from that AST. We briefly summarize the contributions made by the research described in this thesis followed by a discussion of possible enhancements that can be performed in future extensions.

6.1 Contributions

The most significant contribution of our SA is to integrate and utilize state-of-the-art techniques from different research areas such as code reverse engineering, parser generation, and object oriented programming as shown in Chapter 3.

Our new SA introduces a new extensible and flexible algorithm for semantic extraction. Our modifications and upgrades in the algorithm of the SA resulted in a design that is extensible in the following two ways:

1. SA can be easily extended to mine a different programming language. This can be done simply by replacing the abstract syntax tree generator module with another JavaCC produced parser. Using JavaCC parsers not only helps the AST generator to be extensible but also provides a robust parser when the grammar specification file from JavaCC repository is used. This specification was tested over thousands of code files.

2. New algorithms to extract extra information can be added easily. This means when a new heuristic is found, the functionality of this heuristic can be introduced to the SA structure by creating a new visitor pattern class. A visitor pattern that analyzes the AST of the source code can be plugged in and out to enhance the functionality of SA at any time.

Moreover, our new SA algorithm eliminates analyzing unnecessary code. Source code can contain methods that are never used. The functionality inside this unused code

can be misleading. SA extracts the control flow graph of the code and analyzes only the methods that are in the control flow graph.

The design and implementation of new semantic analyzer algorithm improves temporal performance, and also provides the most accurate results as shown in Chapter 5.

Traversing multiple code files to gather the overall semantics is another important feature of our new SA. Since the functionality of the legacy application is usually distributed into separate files, it is important to gather semantics of related modules and be able to combine them to obtain better feedback about the semantics of the application as a whole.

The implementation of SA is done in Java. Java is an object oriented language enabling us to implement object oriented techniques. SA's object oriented structure is another contribution. We see the following benefits of using an object oriented implementation for SA:

- We have the chance of using visitor design patterns. A visitor design pattern is used to separate functionality of information extraction process into separate visitor classes. This makes our code flexible and extensible.
- Object oriented techniques like abstraction and encapsulation are used.Thus, functionality is separated and encapsulated inside separate classes which make its maintenance easier.

SA uses state-of-the-art code reverse engineering techniques to get the source code back from the binary code. If the binary code is available, but the source code is misplaced, SA uses these techniques to get the text of the source code.

6.2 Future Work

In this last section, we discuss possible improvements to SA that can be added to future versions. Our current semantic analyzer extracts semantics from Java code of legacy system. One possible improvement to SA can be functionality to infer semantic

information from the code of stored procedures and triggers as well. Once we analyze the stored procedure calls from the Java code and the functionality of stored procedures, more semantic information will be gathered. Once we analyze the triggers, we can see the effect of *insert*, *update,* and *delete* SQL statements to other entities of the underlying data schema. A trigger is often used to check the constraints of a table. If the constraint is not satisfied with an *insert*, *update,* or *delete* SQL statement, then the trigger cancels the operation. If the constraint is implemented in the code of the trigger instead of implementing in the schema definition of the table, then SE will not discover it, but SA will discover and enhance the extracted schema of the data source. Another usage of the trigger is to make updates, and inserts to related tables if necessary. When we analyze the triggers, we discover relationships between entities that we may not be able to discover with extracting the schema and analyzing the application code.

Operations are recorded in a transaction log in a database system. In database programming, SQL statements that must be executed as a whole are placed in the same transaction and are committed as if they are one atomic statement. Database management systems keep transaction logs to assure that these SQL statements are committed as a one atomic statement. SQL statements in the same transaction are highly related with each other and this relation inherits semantic information. Extending the SA, to analyze the transaction logs, can be another improvement.

Our semantic analyzer identifies its slicing variables according to the user interaction statements inside the code. Then, it investigates the meaning of these slicing variables. Specifically, user interaction statements are input/output statements that are used to interact with the user via the standard input/output. There can be many other

methods of user interaction. Information coming from application can be presented to the user in many ways. We can call the part of the code that presents and gathers information from the user as the *presentation layer*. We can also call the part of the code that inherits the functionality as the *application layer*. The presentation layer of legacy information system can be totally separated from the application layer. This presentation layer can be application specific and can be developed with the help of the technologies like *JSP,* and *HTML.* In such cases information exchanged between the *presentation* and *application layers* inherits semantic information. This information flow between the presentation and application layers can be investigated, and new modules to SA can be added to capture this information. When we analyze this user interaction information and relate it to the underlying schema of the legacy system, more semantics will be gathered.

The number of real application source codes used to test the features of the SA was limited but we have used all the available codes. Future experiments on a large code sample have to be conducted to assert the correctness of our work.

APPENDIX A
GRAMMAR SPECIFICATION FILE EXAMPLE FOR JAVACC

The specification file may start with a section of options to configure the type of parser JavaCC will generate. For example, IGNORE_CASE can be set to force the parser to treat upper and lowercase characters the same.

Then PARSER_BEGIN statement comes. JavaCC uses the string that is passed to this statement to name the parser's class file. After the statement, any Java subroutines can be added in. A main method that instantiates the parser should also be written, so that it can be tested from the command line. This section of the file terminates with a PARSER_END statement.

The last part of the file contains the token definitions and productions.

An example of .jj file is provided in the Figure A-1. The parser that is generated from this grammar specification file reads an Input token. This token consists of one or more Id tokens. The Id tokens consist of a letter, followed by zero or more letters, numbers, or underscores. The parser skips blanks, tabs, carriage returns, and new lines.

Each production has an empty pair of braces at the beginning. This is where Java code is placed for the production (usually variable declarations). Within the production rules, more braces can be added and Java code can be inserted that will execute when that production matches.

When the below specification file is given to JavaCC, these files are created

- IdS.java : main method, methods for each non-terminals, debug(trace) methods, getNetToken method
- IdSConstants.java : interface for tokens

- IdSTokenManager.java : manages the token source file
- ParseException.java : Exceptions
- SimpleCharStream.java : IdSTokenManager uses
- Token.java : describes the input token stream
- TokenMgrError.java : Returns a detailed message for the Error when it is thrown by the token manager to indicate a lexical error.

Table A-1 Example of .jj file.

```
options {
  IGNORE_CASE=true;
  }
PARSER_BEGIN(IdS)
public class IdS {
  public static void main(String args[]) throws ParseException {
    IdS parser = new IdS(System.in);
    parser.Input();
  }
}
PARSER_END(IdS)
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}
TOKEN :
{
  < Id: ["A"-"Z"] ("A"-"Z","0"-"9","_"] )* >
}
void Input() :
{}
{
  ( <Id> )+ <EOF>
}
```

APPENDIX B
COMPILER BASIC TERMS

B.1 Production Rules

Rules in a context free grammar is known as production. Symbols on the left hand sides of the production are known as variables, or non-terminals. Symbols that are to make up the strings derived from the grammar are known as terminals.

B.2 LL and LR Grammars

The two most important classes of grammars are LL and LR. LL stands for left to right, Left –most derivation LR stands for left to right, Right most derivation. In both cases input is read left-to-right. LL parsers are also called top-down or predictive parsers. They construct a parse tree from the root to down. LR parsers are also called bottom-up parsers. They construct a parse tree from leaves up. There are several important subclasses of LR parsers, including SLR, LALR and full LR. LL(2) or LALR(1): number in parenthesis indicates how many tokens of look ahead are required to parse.

B.3 Table Driven Top-Down Parsing

Instead of using its procedure call chain to trace out a tree traversal, the parser maintains a stack containing a list of symbols it expects to see in the future. Initially, the parse stack contains the start symbol of the grammar. When it predicts a production, the parser pops the left-hand side symbol off the stack and pushes the right hand side symbols in reverse order. It also pops the terminal off the stack whenever it matches it against a token from the scanner.

B.4 Lexical Analyzer

Lexical analyzer (‘token manager’ in JavaCC) reads in a sequence of characters and produces a sequence of objects called "tokens". The rules used to break the sequence of characters into a sequence of tokens obviously depend on the language; they are supplied by the programmer as a collection of "regular expressions".

B.5 The Parser

The parser consumes the sequence of tokens, analyses its structure, and produces "abstract syntax tree".

APPENDIX C
DECOMPILED FILE EXAMPLE

In this Appendix, we give an example to show how accurately Jad decompiler can convert *.class* files to *.java* files. In Table C.1, we show the original source code of *ScheduleMenu.java*. We compile *ScheduleMenu.java* and get the *ScheduleMenu.class* file. Then, we de-compile *ScheduleMenu.class* file with Jad decompiler and get another version of *ScheduleMenu.java* code. In Table C.2, we show the de-compiled version of *ScheduleMenu.java*.

Table C.1 Original version of ScheduleMenu.java code.

```
import java.sql.*;
import java.util.*;
import java.io.*;
public class ScheduleMenu{
        static String database = "jdbc:oracle:thin:@oracle.cise.ufl.edu:1521:oradb";
        static String user = "****"; static String password = "******";
        static int totalActivityNumber;
        static Connection jdbcConnection;    static Statement sqlStatement;
        static ResultSet resultSet=null;         static String query = null;
public static void doDatabaseInitializations(String databaseName, String userName,
String pass)  throws Exception          {
/******* CONNECTS TO DATABASE *******/
Class.forName("oracle.jdbc.driver.OracleDriver");
jdbcConnection = DriverManager.getConnection(databaseName,userName,pass);
/******* PREPARES SQL STATEMENT *******/
sqlStatement = jdbcConnection.createStatement();
}
public static String readLineFromConsole()
{
        String lineRead = "";
        try {
        BufferedReader br = null;
        InputStreamReader in1 = null;
        in1 = new InputStreamReader(System.in);
        br = new BufferedReader(in1);
        lineRead = br.readLine();
```

```
        lineRead = lineRead.trim();
        }catch(Exception e){
                System.out.println(e.getMessage());
        }
        return lineRead;
}
public static int getTotalActivityNumber()  throws Exception        {
 /******* GET TOTAL ACTIVITY NUMBERS *******/
 query = "SELECT Count(Activ_no) FROM Activity ";
 resultSet = sqlStatement.executeQuery(query);
 int activityCount = 0;
 while(resultSet.next())
 {
        activityCount = resultSet.getInt(1);
 }
resultSet.close();
return activityCount;
}
public static String menu()     {
        System.out.println("****************************************");
        System.out.println("Please make a selection:");
        System.out.println("1. List all the projects");
        System.out.println("2. List the activities");
        System.out.println("3. Select an activity to reschedule into the first available
slot");
        System.out.println("4. Select an activity to shift");
        System.out.println("5. Select an activity to change its place");
        System.out.println("6. Exit Application");
        System.out.println("****************************************");
        System.out.println("Please enter the option number you selected: ");
        return readLineFromConsole();
}
        public static void main(String[] args) throws Exception{
                String menuOption="*";
                doDatabaseInitializations(database,user,password);
                                System.out.println(menuOption);
                totalActivityNumber = getTotalActivityNumber();
                menuOption = "1";
                ScheduleList scheduleList = new ScheduleList();
                ManipulateActivity manipulateSchedule = new ManipulateActivity();
                while (! menuOption.equals("6"))
                {
                        menuOption = menu();
                        if (menuOption.equals("1"))
                        {
                                scheduleList.listProjectNames(sqlStatement,resultSet);
```

```
//list project names
                        }
                        if (menuOption.equals("2"))
                        {
                                scheduleList.listActivities(sqlStatement,resultSet); //list
activities
                        }
                        if (menuOption.equals("3"))
                        {

        manipulateSchedule.rescheduleActivity(sqlStatement,resultSet,jdbcConnection);
                        //"(3) Select an activity to reschedule into the first available slot"
                        }
                        if (menuOption.equals("4"))
                        {

        manipulateSchedule.shiftActivity(sqlStatement,resultSet,jdbcConnection);
                        //""(4) Select an activity to shift""
                        }
                        if (menuOption.equals("5"))
                        {

        manipulateSchedule.changeActivity(sqlStatement,resultSet,jdbcConnection);
                        //""(4) Select an activity to change place""
                        }
                }
        }
        //end main }
```

Table C.2 De-compiled version of ScheduleMenu.java code.

```
// Decompiled by Jad v1.5.8e2. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://kpdus.tripod.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   ScheduleMenu.java

import java.io.*;
import java.sql.*;
public class ScheduleMenu
{
   public ScheduleMenu()
   {
   }
   public static void doDatabaseInitializations(String s, String s1, String s2)   throws
Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
    jdbcConnection = DriverManager.getConnection(s, s1, s2);
    sqlStatement = jdbcConnection.createStatement();
  }
  public static String readLineFromConsole()
  {
    String s = "";
    try
    {
      BufferedReader bufferedreader = null;
      InputStreamReader inputstreamreader = null;
      inputstreamreader = new InputStreamReader(System.in);
      bufferedreader = new BufferedReader(inputstreamreader);
      s = bufferedreader.readLine();
      s = s.trim();
    }
    catch(Exception exception)
    {
      System.out.println(exception.getMessage());
    }
    return s;
  }
  public static int getTotalActivityNumber()   throws Exception
  {
    query = "SELECT Count(Activ_no) FROM Activity ";
    resultSet = sqlStatement.executeQuery(query);
    int i;
    for(i = 0; resultSet.next(); i = resultSet.getInt(1));
    resultSet.close();
    return i;
  }
  public static String menu()
  {
    System.out.println("****************************************");
    System.out.println("Please make a selection:");
    System.out.println("1. List all the projects");
    System.out.println("2. List the activities");
    System.out.println("3. Select an activity to reschedule into the first available slot");
    System.out.println("4. Select an activity to shift");
    System.out.println("5. Select an activity to change its place");
    System.out.println("6. Exit Application");
    System.out.println("****************************************");
    System.out.println("Please enter the option number you selected: ");
    return readLineFromConsole();
  }
  public static void main(String args[])  throws Exception
  {
```

```
        String s = "*";
        doDatabaseInitializations(database, user, password);
        System.out.println(s);
        totalActivityNumber = getTotalActivityNumber();
        s = "1";
        ScheduleList schedulelist = new ScheduleList();
        ManipulateActivity manipulateactivity = new ManipulateActivity();
        while(!s.equals("6"))
        {
           s = menu();
           if(s.equals("1"))
              ScheduleList.listProjectNames(sqlStatement, resultSet);
           if(s.equals("2"))
              ScheduleList.listActivities(sqlStatement, resultSet);
           if(s.equals("3"))
              manipulateactivity.rescheduleActivity(sqlStatement, resultSet,
jdbcConnection);
           if(s.equals("4"))
              manipulateactivity.shiftActivity(sqlStatement, resultSet, jdbcConnection);
           if(s.equals("5"))
              ManipulateActivity.changeActivity(sqlStatement, resultSet, jdbcConnection);
        }
     }
     static String database = "jdbc:oracle:thin:@oracle.cise.ufl.edu:1521:oradb";
     static String user = "****";
     static String password = "******";
     static int totalActivityNumber;
     static Connection jdbcConnection;
     static Statement sqlStatement;
     static ResultSet resultSet = null;
     static String query = null;
}
```

APPENDIX D
VISITOR CLASS EXAMPLE

In this example, the implementation of a visitor class on a heterogeneous aggregate structure will be seen. This structure defines the first production rules of java grammar. Compilation unit has Package Declaration, Import Declaration, Type Declaration and <EOF> in its production rule. Several production rules of Java grammar can be seen in the Figure D-1.

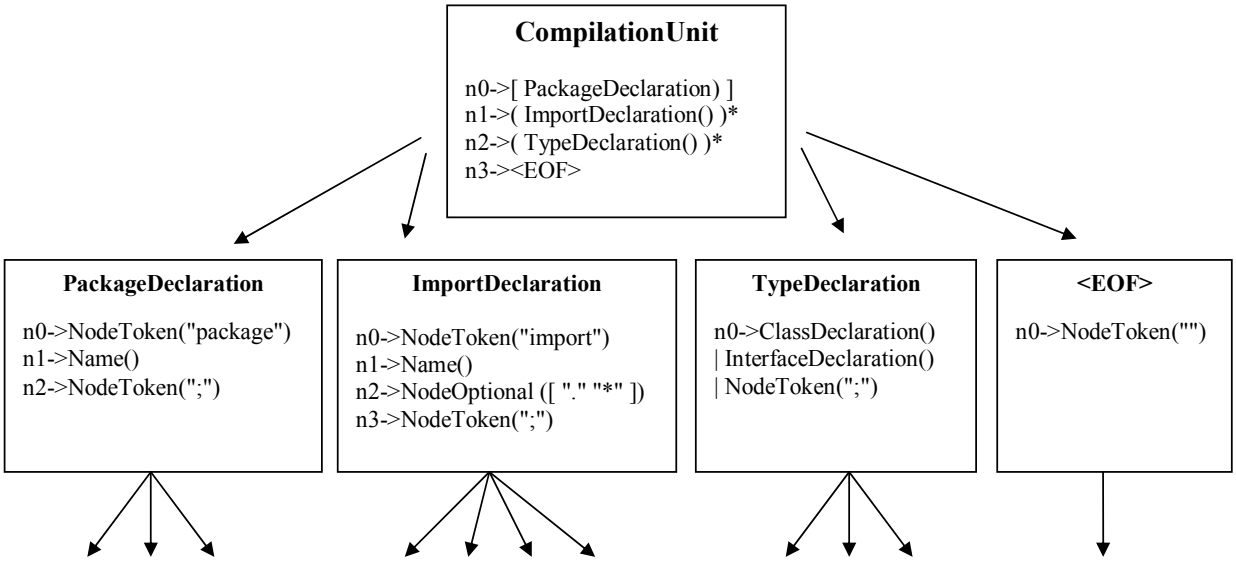


Figure D-1 Example of production rules of Java grammar.

CompilationUnit, PackageDeclaration, TypeDeclaration class definition of this structure can be seen in the Table D.1. Each class has an accept method. This method returns the control back to the visitor class by calling it by giving itself as a parameter.

Table D.1 Class Definitions of Example Nodes of Tree

```
public class CompilationUnit implements Node {
  public NodeOptional f0;
  public NodeListOptional f1;
  public NodeListOptional f2;
  public NodeToken f3;
  public Object accept(visitor.ObjectVisitor v, Object argu) {
    return v.visit(this,argu);
  }
}

public class PackageDeclaration implements Node {
  public NodeToken f0;
  public Name f1;
  public NodeToken f2;
  public Object accept(visitor.ObjectVisitor v, Object argu) {
    return v.visit(this,argu);
  }
}

public class TypeDeclaration implements Node {
  public NodeChoice f0;
public Object accept(visitor.ObjectVisitor v, Object argu) {
    return v.visit(this,argu);
  }
}
```

The visitor interface has visit method for each of the node in the tree. The visitor class that implements this interface has to write a method to perform functionality for each of this node.

Table D.2 Visitor Interface

```
Public interface ObjectVisitor {
  public Object visit(CompilationUnit n, Object argu);
  public Object visit(PackageDeclaration n, Object argu);
  public Object visit(ImportDeclaration n, Object argu);
  public Object visit(TypeDeclaration n, Object argu);
  public Object visit(ClassDeclaration n, Object argu);
…
}
```

In the table below, a visitor class can be examined. Visitor class implements the interface ObjectVisitor and writes the desired functionality inside the appropriate methods. The visitor ObjectDepthFirst only traverses the tree in dept first order. The comments above each visit method shows which field corresponds to which part of the production.

Table D.3 Visitor Class Example

```
public class ObjectDepthFirst implements ObjectVisitor {
  /**
   * f0 -> [ PackageDeclaration() ]
   * f1 -> ( ImportDeclaration() )*
   * f2 -> ( TypeDeclaration() )*
   * f3 -> <EOF>
   */
  public Object visit(CompilationUnit n, Object argu) {
    Object _ret=null;
    n.f0.accept(this, argu);
    n.f1.accept(this, argu);
    n.f2.accept(this, argu);
    n.f3.accept(this, argu);
    return _ret;
  }

  /**
   * f0 -> "package"
   * f1 -> Name()
   * f2 -> ";"
   */
  public Object visit(PackageDeclaration n, Object argu) {
    Object _ret=null;
    n.f0.accept(this, argu);
    n.f1.accept(this, argu);
    n.f2.accept(this, argu);
    return _ret;
  }

  /**
   * f0 -> "import"
   * f1 -> Name()
   * f2 -> [ "." "*" ]
   * f3 -> ";"
   */
```

```
   public Object visit(ImportDeclaration n, Object argu) {
     Object _ret=null;
     n.f0.accept(this, argu);
     n.f1.accept(this, argu);
     n.f2.accept(this, argu);
     n.f3.accept(this, argu);
     return _ret;
   }

   /**
    * f0 -> ClassDeclaration()
    *       | InterfaceDeclaration()
    *       | ";"
    */
   public Object visit(TypeDeclaration n, Object argu) {
     Object _ret=null;
     n.f0.accept(this, argu);
     return _ret;
   }
}
```

APPENDIX E
JAVA GRAMMAR

Java Grammar production rules can be seen below.[1]

```
CompilationUnit = [ PackageDeclaration() ] ( ImportDeclaration() )* ( TypeDeclaration() )* <EOF>

PackageDeclaration = "package" Name() ";" "import" Name() [ "." "*" ]          ";" ImportDeclaration

TypeDeclaration = ClassDeclaration()*
                | InterfaceDeclaration()*
                | ";"

ClassDeclaration = ( "abstract" | "final" | "public" | "strictfp" )* UnmodifiedClassDeclaration()

UnmodifiedClassDeclaration = "class" <IDENTIFIER> [ "extends" Name() ] [ "implements" NameList() ]
ClassBody()

ClassBody = "{"  ( ClassBodyDeclaration() )*          "}"

NestedClassDeclaration = ( "static" | "abstract" | "final" | "public" | "protected" | "private" | "strictfp" )*
        UnmodifiedClassDeclaration()

ClassBodyDeclaration = Initializer()
                     | NestedClassDeclaration()
                     | NestedInterfaceDeclaration()
                     | ConstructorDeclaration()
                     | MethodDeclaration()
                     | FieldDeclaration()

MethodDeclarationLookahead = ( "public" | "protected" | "private" | "static" | "abstract" | "final" | "native" |
"synchronized" | "strictfp" )* ResultType() <IDENTIFIER> "("

InterfaceDeclaration = ( "abstract" | "public" | "strictfp" )* UnmodifiedInterfaceDeclaration()

NestedInterfaceDeclaration =  ( "static" | "abstract" | "final" | "public" | "protected" | "private" | "strictfp" )*
UnmodifiedInterfaceDeclaration()

UnmodifiedInterfaceDeclaration =  "interface" <IDENTIFIER> [ "extends" NameList() ] "{" (
InterfaceMemberDeclaration() )* "}"

InterfaceMemberDeclaration = NestedClassDeclaration()
                           | NestedInterfaceDeclaration()
                           | MethodDeclaration()
                           | FieldDeclaration()
```

[1] Java Grammar is extracted from the .jj file of Java Grammar that is obtained from JavaCC Repository [6].

```
FieldDeclaration = ( "public" | "protected" | "private" | "static" | "final" | "transient" | "volatile" )* Type()
VariableDeclarator() ( "," VariableDeclarator() )* ";"

VariableDeclarator = VariableDeclaratorId() [ "=" VariableInitializer() ]

VariableDeclaratorId =  <IDENTIFIER> ( "[" "]" )*

VariableInitializer =  ArrayInitializer() | Expression()

ArrayInitializer = "{" [ VariableInitializer() ( "," VariableInitializer() )* ] [ "," ] "}"

MethodDeclaration = ( "public" | "protected" | "private" | "static" | "abstract" | "final" | "native" |
"synchronized" | "strictfp" )* ResultType() MethodDeclarator() [ "throws" NameList() ] ( Block() | ";" )

MethodDeclarator =   <IDENTIFIER> FormalParameters() ( "[" "]" )*

FormalParameters =  "(" [ FormalParameter() ( "," FormalParameter() )* ] ")"

FormalParameter =  [ "final" ] Type() VariableDeclaratorId()

ConstructorDeclaration = [ "public" | "protected" | "private" ] <IDENTIFIER> FormalParameters() [
"throws" NameList() ] "{" [ ExplicitConstructorInvocation() ] ( BlockStatement() )* "}"

ExplicitConstructorInvocation =  "this" Arguments() ";"
                                    | [ PrimaryExpression() "." ] "super" Arguments() ";"

Initializer =  [ "static" ] Block()

Type =  ( PrimitiveType() | Name() ) ( "[" "]" )*

PrimitiveType = "boolean"
                    | "char"
                    | "byte"
                    | "short"
                    | "int"
                    | "long"
                    | "float"
                    | "double"

ResultType =  "void"
                    | Type()

Name =  <IDENTIFIER> ( "." <IDENTIFIER> )*

NameList =  Name() ( "," Name() )*

Expression =  ConditionalExpression() [ AssignmentOperator() Expression() ]

AssignmentOperator =  "="
                    | "*="
                    | "/="
                    | "%="
                    | "+="
                    | "-="
                    | "<<="
                    | ">>="
```

```
                | ">>>="
                | "&="
                | "^="
                | "|="

ConditionalExpression =  ConditionalOrExpression() [ "?" Expression() ":" ConditionalExpression() ]

ConditionalOrExpression =  ConditionalAndExpression() ( "||" ConditionalAndExpression() )*

ConditionalAndExpression = InclusiveOrExpression() ( "&&" InclusiveOrExpression() )*

InclusiveOrExpression =  ExclusiveOrExpression() ( "|" ExclusiveOrExpression() )*

ExclusiveOrExpression =  AndExpression() ( "^" AndExpression() )*

AndExpression =  EqualityExpression() ( "&" EqualityExpression() )*

EqualityExpression =  InstanceOfExpression() ( ( "==" | "!=" ) InstanceOfExpression() )*

InstanceOfExpression =  RelationalExpression() [ "instanceof" Type() ]

RelationalExpression =  ShiftExpression() ( ( "<" | ">" | "<=" | ">=" ) ShiftExpression() )*

ShiftExpression =  AdditiveExpression() ( ( "<<" | ">>" | ">>>" ) AdditiveExpression() )*

AdditiveExpression =  MultiplicativeExpression() ( ( "+" | "-" ) MultiplicativeExpression() )*

MultiplicativeExpression =  UnaryExpression() ( ( "*" | "/" | "%" ) UnaryExpression() )*

UnaryExpression = ( "+" | "-" ) UnaryExpression()
                | PreIncrementExpression()
                | PreDecrementExpression()
                | UnaryExpressionNotPlusMinus()

PreIncrementExpression = "++" PrimaryExpression()

PreDecrementExpression = "--" PrimaryExpression()

UnaryExpressionNotPlusMinus =  ( "~" | "!" ) UnaryExpression()
                             | CastExpression()
                             | PostfixExpression()

CastLookahead = "(" PrimitiveType()
                | "(" Name() "[" "]"
                | "(" Name() ")" ( "~" | "!" | "(" | <IDENTIFIER> | "this" | "super" | "new" | Literal() )

PostfixExpression =  PrimaryExpression() [ "++" | "--" ]

CastExpression =   "(" Type() ")" UnaryExpression()
                | "(" Type() ")" UnaryExpressionNotPlusMinus()

PrimaryExpression =  PrimaryPrefix() ( PrimarySuffix() )*

PrimaryPrefix = Literal()
                | "this"
                | "super" "." <IDENTIFIER>
```

```
                    | "(" Expression() ")"
                    | AllocationExpression()
                    | ResultType() "." "class"
                    | Name()

PrimarySuffix = "." "this"
                    | "." AllocationExpression()
                    | "[" Expression() "]"
                    | "." <IDENTIFIER>
                    | Arguments()

Literal = <INTEGER_LITERAL>
                    | <FLOATING_POINT_LITERAL>
                    | <CHARACTER_LITERAL>
                    | <STRING_LITERAL>
                    | BooleanLiteral()
                    | NullLiteral()

BooleanLiteral = "true"
                    | "false"

NullLiteral =  "null"

Arguments = "(" [ ArgumentList() ] ")"

ArgumentList = Expression() ( "," Expression() )*

AllocationExpression = "new" PrimitiveType() ArrayDimsAndInits()
                    | "new" Name() ( ArrayDimsAndInits() | Arguments() [ ClassBody() ] )


ArrayDimsAndInits = ( "[" Expression() "]" )+ ( "[" "]" )*
                    | ( "[" "]" )+ ArrayInitializer()

Statement = LabeledStatement()
          | Block()
          | EmptyStatement()
          | StatementExpression() ";"
          | SwitchStatement()
          | IfStatement()
          | WhileStatement()
          | DoStatement()
          | ForStatement()
          | BreakStatement()
          | ContinueStatement()
          | ReturnStatement()
          | ThrowStatement()
          | SynchronizedStatement()
          | TryStatement()
          | AssertStatement()

LabeledStatement = <IDENTIFIER> ":" Statement()

Block = "{" ( BlockStatement() )* "}"

BlockStatement = LocalVariableDeclaration() ";"
```

```
                | Statement()
                | UnmodifiedClassDeclaration()
                | UnmodifiedInterfaceDeclaration()

LocalVariableDeclaration = [ "final" ] Type() VariableDeclarator() ( "," VariableDeclarator() )*

EmptyStatement = ";"

StatementExpression = PreIncrementExpression()
                | PreDecrementExpression()
                | PrimaryExpression() [ "++" | "--" | AssignmentOperator() Expression() ]

SwitchStatement = "switch" "(" Expression() ")" "{" ( SwitchLabel() ( BlockStatement() )* )* "}"

SwitchLabel = "case" Expression() ":"
                | "default" ":"

IfStatement = "if" "(" Expression() ")" Statement() [ "else" Statement() ]

WhileStatement = "while" "(" Expression() ")" Statement()

DoStatement = "do" Statement() "while" "(" Expression() ")" ";"

ForStatement = "for" "(" [ ForInit() ] ";" [ Expression() ] ";" [ ForUpdate() ] ")" Statement()

ForInit = LocalVariableDeclaration()
        | StatementExpressionList()

StatementExpressionList =  StatementExpression() ( "," StatementExpression() )*

ForUpdate =  StatementExpressionList()

BreakStatement =  "break" [ <IDENTIFIER> ] ";"

ContinueStatement =  "continue" [ <IDENTIFIER> ] ";"

ReturnStatement =  "return" [ Expression() ] ";"

ThrowStatement =  "throw" Expression() ";"

SynchronizedStatement =  "synchronized" "(" Expression() ")" Block()

TryStatement =  "try" Block() ( "catch" "(" FormalParameter() ")" Block() )* [ "finally" Block() ]

AssertStatement = "assert" Expression() [ ":" Expression() ] ";
```

APPENDIX F
ORACLE SQL GRAMMAR

Oracle SQL Grammar production rules can be seen below.[1]

```
SQLAndExpr = SQLNotExpr() ( <AND> SQLNotExpr() )*

SQLBetweenClause = [ <NOT> ] <BETWEEN> SQLSumExpr() <AND> SQLSumExpr()

SQLColRef =  SQLLvalue()

SQLCompareExpr =  ( SQLSelect() | SQLIsClause() | SQLExistsClause() | SQLSumExpr() [
SQLCompareExprRight() ] )

SQLCompareExprRight =  ( SQLLikeClause() | SQLInClause() | SQLLeftJoinClause() |
SQLRightJoinClause() | SQLBetweenClause() | SQLCompareOp() SQLSumExpr() )

SQLCompareOp =  ( <EQUAL> | <NOTEQUAL> | <NOTEQUAL2> | <GREATER> |
<GREATEREQUAL> | <LESS> | <LESSEQUAL> )

SQLCursorArgs = "(" [ ( SQLColRef() [ SQLDataType() ] | SQLFunction() ) ( "," ( SQLColRef() [
SQLDataType() ] | SQLFunction() ) )* ] ")"

SQLCursorClose = <CLOSE> <ID>

SQLCursorDeclare = "cursor" <ID> [ SQLCursorArgs() ] "is" SQLSelect() ";"

SQLCursorFetch = <FETCH> SQLSelectCols() [ <INTO> SQLSelectCols() ]

SQLCursorOpen = <OPEN> <ID> [ SQLCursorArgs() ]

SQLDataType = "integer"
                | "smallint"
                | "number" [ "(" <INTEGER_LITERAL> ")" ]
                | ( "char" | "character" ) [ "(" <INTEGER_LITERAL> ")" ]
                | "varchar2" [ "(" <INTEGER_LITERAL> ")" ]
                | "boolean"

SQLDelete = <DELETE> <FROM> SQLTableList() [ SQLWhere() ]

SQLExistsClause = <EXISTS> "(" SQLSelect() ")"

SQLFunction = ( <UPPER> SQLFunctionArgs() | <MAX> SQLFunctionArgs() | <MIN>
SQLFunctionArgs() | <SUM> SQLFunctionArgs() | <COUNT> SQLFunctionArgs() | <LPAD>
SQLFunctionArgs() | <LTRIM> SQLFunctionArgs() | <RTRIM> SQLFunctionArgs() | <LENGTH>
```

[1] Oracle SQL Grammar is extracted from the .jj file of Oracle SQL Grammar that is obtained from JavaCC Repository.

```
SQLFunctionArgs() | <REPLACE> SQLFunctionArgs() | <SUBSTR> SQLFunctionArgs() | <TO_CHAR>
SQLFunctionArgs() | <TO_NUMBER> SQLFunctionArgs() | <ORIGINPLUS> <DOT>
<CONVERT_TIMESTAMP_TO_DATE> SQLFunctionArgs() | <ORIGINPLUS> <DOT>
<FORMAT_AUDIT_HEADER> SQLFunctionArgs() | <ORIGINPLUS> <DOT>
<FORMAT_ATTRIBUTE_SUBSTRING> SQLFunctionArgs() | <ORIGINPLUS> <DOT>
<GET_CURRENT_TIMESTAMP> | <ID> SQLFunctionArgs() )

SQLFunctionArgs = "(" [ SQLSumExpr() ( "," SQLSumExpr() )* ]  ")"

SQLGroupBy = <GROUP> <BY> SQLOrderByList()

SQLInClause = [ <NOT> ] <IN> "(" SQLLValueList() ")"

SQLInsert = <INSERT> <INTO> SQLTableList() [ "(" SQLSelectCols() ")" <VALUES> ] "("
SQLSelectCols() ")"

SQLIsClause =  SQLColRef() <IS> [ <NOT> ] <NULL>

SQLLeftJoinClause = "(+)" SQLCompareOp() SQLSumExpr()

SQLLikeClause = [ <NOT> ] <LIKE> SQLPattern()

SQLLiteral = ( <STRING_LITERAL> | <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | (
<ZEROS> | <ZERO> ) | <SPACES> | <ASTERISK> )

SQLLvalue = ( <REQUEST> <DOT> SQLLvalueTerm() | <REPLY_REPEATING_GROUP> <DOT>
SQLLvalueTerm() | <REPLY> <DOT> SQLLvalueTerm() | <SQL_I_O_CORRECT> |
<ROW_NOT_FOUND> | SQLLvalueTerm() )

SQLLvalueTerm = <ID> ( <DOT> <ID> )*

SQLNotExpr = [ <NOT> ] SQLCompareExpr()

SQLOrderBy = <ORDER> <BY> SQLOrderByList()

SQLOrderByElem = SQLColRef() [ SQLOrderDirection() ]

SQLOrderByList = SQLOrderByElem() ( "," SQLOrderByElem() )*

SQLOrderDirection = ( <ASC> | <DESC> )

SQLOrExpr =  SQLAndExpr()     ( <OR> SQLAndExpr() )*

SQLPattern =     ( <STRING_LITERAL> | "?" | <USER> | SQLLvalue() )

SQLProductExpr =  SQLUnaryExpr() ( ( "*" | "/" ) SQLUnaryExpr() )*

SQLRightJoinClause =  SQLCompareOp() SQLSumExpr() "(+)"

SQLSelect = <SELECT> SQLSelectCols() [ <INTO> SQLSelectCols() ] <FROM> SQLTableList() [
SQLWhere() ] [ SQLGroupBy() ] [ SQLOrderBy() ]

SQLSelectCols = ( <ALL> | <DISTINCT> )* [ "*" | SQLSelectList() ]

SQLSelectList =  SQLSumExpr() ( "," SQLSumExpr() )*
```

SQLStatement = (SQLSelect() | SQLInsert() | SQLUpdate() | SQLDelete() | SQLCursorOpen() | SQLCursorFetch() | SQLCursorClose()) ";"

SQLSumExpr = SQLProductExpr() (("+" | "-" | "||") SQLProductExpr())*

SQLTableList = SQLTableRef() ("," SQLTableRef())*

SQLTableRef = <ID> [<ID>]

SQLTerm = ("(" SQLOrExpr() ")" | SQLColRef() | SQLLiteral() | SQLFunction())

SQLUnaryExpr = [("+" | "-")] SQLTerm()

SQLUpdate = <UPDATE> SQLTableList() <SET> (SQLUpdateAssignment() [","])+ [SQLWhere()]

SQLUpdateAssignment = SQLLvalue() "=" (SQLTerm() (<CONCAT> SQLTerm())+ | SQLSumExpr())

SQLLValueElement = (<NULL> | SQLSumExpr() | SQLSelect())

SQLLValueList = SQLLValueElement() ("," SQLLValueElement())*

SQLWhere = <WHERE> SQLOrExpr()

APPENDIX G
CONSTRUCTION SCHEDULING PROGRAM

This program does a very simple re-scheduling of an activity of a construction firm. The functionality of scheduling program is separated into three files, *ScheduleMenu.java, ScheduleList.java,* and *ManipulateActivity.java*. The content of these files can be seen below. The program has the following functionalities:

- Lists the projects.
- Lists the activities.
- Reschedules the activity into the first available slot.
- Shifts the start and finish date of the activity as much as possible.
- Changes the order of the activities in the schedule.

Selection of the desired functionality is done in *ScheduleMenu.java* file. *ScheduleList.java* file lists the projects and activities. *ManipulateActivity.java* file has the functionality for rescheduling the activity, shifting the date, and changing the order of the activity. ScheduleMenu.java, ScheduleList.java, and ManipulateActivity.java are shown in Table G.1, Table G.2, and Table G.3 respectively.

Table G.1 Code of ScheduleMenu.java.

```
import java.sql.*;
import java.util.*;
import java.io.*;

public class ScheduleMenu{
static String database = "jdbc:oracle:thin:@oracle.cise.ufl.edu:1521:oradb";
static String user = "seek";static String password = "seek2002";
static Connection jdbcConnection;static Statement sqlStatement;
static ResultSet resultSet=null;static String query = null;static int totalActivityNumber;

public static void doDatabaseInitializations(String databaseName, String userName,
String pass)  throws Exception{
  Class.forName("oracle.jdbc.driver.OracleDriver");
  jdbcConnection = DriverManager.getConnection(databaseName,userName,pass);
```

```
  sqlStatement = jdbcConnection.createStatement();
          }

public static int getTotalActivityNumber()  throws Exception{
  query = "SELECT Count(Activ_no) FROM Activity ";
  resultSet = sqlStatement.executeQuery(query); int activityCount = 0;
  while(resultSet.next()){
          activityCount = resultSet.getInt(1);
  }
  resultSet.close();
  return activityCount;
}

public static String menu()
{
          System.out.println("****************************************");
          System.out.println("Please make a selection:");
          System.out.println("1. List all the projects");
          System.out.println("2. List the activities");
          System.out.println("3. Select an activity to reschedule ");
          System.out.println("4. Select an activity to shift");
          System.out.println("5. Select an activity to change its place");
          System.out.println("6. Exit Application");
          System.out.println("****************************************");
          System.out.println("Please enter the option number you selected: ");
          String input = "";
    try {
          InputStreamReader in1 = new InputStreamReader(System.in);
          BufferedReader br = new BufferedReader(in1);
          input = br.readLine();
          }catch(Exception e){
          System.out.println(e.getMessage());
          }
return input;
}

public static void main(String[] args) throws Exception       {
          String menuOption="*";                  menuOption = "1";
          doDatabaseInitializations(database,user,password);
          totalActivityNumber = getTotalActivityNumber();
          ScheduleList scheduleList = new ScheduleList();
          ManipulateActivity manipulateSchedule = new ManipulateActivity();
          while (! menuOption.equals("6")){
                    menuOption = menu();
                    //list project names
                    if (menuOption.equals("1")){
```

```
                        scheduleList.initiliazeProjectQuery(sqlStatement,resultSet);
                }
                //list activities
                if (menuOption.equals("2")){
                        scheduleList.listActivities(sqlStatement,resultSet);
                }
                //(3) Select an activity to reschedule into the first available slot
                if (menuOption.equals("3")){
        manipulateSchedule.rescheduleActivity(sqlStatement,resultSet,jdbcConnection);
                }
                //(4) Select an activity to shift
                if (menuOption.equals("4")){
        manipulateSchedule.shiftActivity(sqlStatement,resultSet,jdbcConnection);
                }
                //(5) Select an activity to change place
                if (menuOption.equals("5")){
        manipulateSchedule.changeActivity(sqlStatement,resultSet,jdbcConnection);
                }
        }
}
}
```

Table G.2 Code of ScheduleList.java.

```
import java.sql.*;
import java.util.*;
import java.io.*;
public class ScheduleList{
public static void initiliazeProjectQuery(Statement sqlStatement,ResultSet resultSet){
 String query = "SELECT p_id FROM Project";
 resultSet = sqlStatement.executeQuery(query);
  listProjects(sqlStatement,resultSet);
}
public static void listProjects(Statement sqlStatement,ResultSet resultSet)  {
        String projNumber = null;
        System.out.println("=======================================");
        System.out.println("The following are all the projects found:");
        while(resultSet.next()){
                projNumber = resultSet.getString(1);
                System.out.println("Project Num:" + projNumber);
        }
        resultSet.close();
}
public static void listActivities(Statement sqlStatement,ResultSet resultSet){
        String currentactivity = null;  String projectNo = null;  String workItemNo = null;
        String query = null; String finishdate = null;  String startdate = null;  int dur = 0;
        query = "SELECT Activ_no, activ_planned_startdate, activ_planned_finishdate,
```

```
        Workitem_no, Proj_no FROM Activity ORDER BY activ_planned_startdate";
        resultSet = sqlStatement.executeQuery(query); int totalActivityNumber = 0;
        System.out.println("proj_no workitemno activityno startdate finishdate ");
        System.out.println("-------------------------------------------------");
        while(resultSet.next())
        {
                currentactivity = resultSet.getString(1);
                startdate = resultSet.getString(2);
                startdate = startdate.substring(0,10);
                finishdate = resultSet.getString(3);
                finishdate = finishdate.substring(0,10);
                workItemNo = resultSet.getString(4);
                projectNo = resultSet.getString(5);
                totalActivityNumber = totalActivityNumber + 1;
                System.out.println(projectNo);
                System.out.println("\t" + workItemNo);
                System.out.println("\t" + currentactivity);
                System.out.println("\t" + startdate);
                System.out.println("\t" + finishdate );
         }
        resultSet.close();
        System.out.println("Number of activities: " + totalActivityNumber);
        System.out.println("=========================================");
}}
```

Table G.3 Code of ManipulateActivity.java.

```
import java.sql.*;
import java.util.*;
import java.io.*;
public class ManipulateActivity
{
public void rescheduleActivity(Statement sqlStatement,ResultSet resultSet,Connection
jdbcConnection)
{
  String inputActivityNo = null; String currentActivity = null;
  String successorActivity = null; String query = null;
  boolean noRecord = false;  boolean slotFound = false;
  String oldStartDate = null;  String finishDate = null;
  String startDate = null; startDateSQL = null;
  java.sql.Date finishDateSQL = null; java.sql.Date  int duration = 0;
  int dateDifference = 0; String firstLink = null; String lastLink = null;
  String lineRead = "";
  System.out.println("=========================================");
  System.out.println("Please type which activity you would like to reschedule: ");
try {
    InputStreamReader in1 = new InputStreamReader(System.in);
```

```
    BufferedReader br = new BufferedReader(in1);
    inputActivityNo = br.readLine();
}catch(Exception e){
          System.out.println(e.getMessage());
}
if (inputActivityNo.equals("P001")) {
  System.out.println("First Activity P001 can not be rescheduled ...");
}else if (inputActivityNo.equals("R001")){
  System.out.println("Last Activity R001 can not be rescheduled ...");
}else{
  System.out.println("Activity is being rescheduled to the earliest date possible...");
  String query1; query1 = "SELECT activ_duration,activ_planned_startdate " ;
  String query2; query2 = " FROM Activity WHERE Activ_no = '"+ inputActivityNo +
"'";
  query = query1 + query2; noRecord = true;
  resultSet = sqlStatement.executeQuery(query);
  while (resultSet.next())
{
    duration = resultSet.getInt(1);
    oldStartDate = resultSet.getString(2);
    oldStartDate = oldStartDate.substring(0,10);
   noRecord = false;
}
resultSet.close();
if (noRecord){
          System.out.println(inputActivityNo + " activity could not be found ");
}else{
          System.out.println("---------------------------------------");
          System.out.println("activity no: " + inputActivityNo);
          System.out.println("old start date: "  + oldStartDate);
          System.out.println("duration: " + duration);
          currentActivity = "P001";
while ((! inputActivityNo.equals(currentActivity)) && (! slotFound))
{
          query = "SELECT (B.Activ_planned_startdate - A.activ_planned_finishdate)
FROM ACTIVITY A, ACTIVITY B WHERE A.activ_successor = B.activ_no AND
A.activ_no = '" + currentActivity + "'";
          resultSet = sqlStatement.executeQuery(query);
          resultSet.next();
          dateDifference = resultSet.getInt(1);
          resultSet.close();
          if(dateDifference >= duration){
                    slotFound = true;
                    query = "SELECT Activ_successor FROM Activity WHERE activ_no=
'"+ currentActivity + "'";
                    resultSet = sqlStatement.executeQuery(query);
```

```
                    resultSet.next();
                    successorActivity = resultSet.getString(1);
                    resultSet.close();
                    /* get firstlink and lastlink */
                    query = "SELECT activ_predecessor, activ_successor FROM Activity
WHERE activ_no = '" + inputActivityNo + "'";
                    resultSet = sqlStatement.executeQuery(query);
                    resultSet.next();
                    firstLink = resultSet.getString(1);
                    lastLink = resultSet.getString(2);
                    resultSet.close();
                    String preAct = firstLink;
                    String sucAct = lastLink;
                    sqlStatement.executeUpdate("UPDATE Activity SET activ_successor = '"
+ inputActivityNo + "' WHERE activ_no = '" + currentActivity + "'");
                    sqlStatement.executeUpdate("UPDATE Activity SET activ_predecessor =
'" + inputActivityNo + "'        WHERE activ_no = '" + successorActivity + "'");
                    sqlStatement.executeUpdate("UPDATE Activity SET activ_successor = '"
+ sucAct + "' WHERE activ_no = '" + firstLink + "'");
                    sqlStatement.executeUpdate("UPDATE Activity SET activ_predecessor =
'"  + preAct + "' WHERE activ_no = '" + lastLink + "'");
                    query = "SELECT (activ_planned_finishdate + 1) FROM Activity
WHERE activ_no = '" + currentActivity + "'";
                    resultSet = sqlStatement.executeQuery(query);
                    resultSet.next();
                    startDateSQL = resultSet.getDate(1);
                    startDate = resultSet.getString(1);
                    startDate = startDate.substring(0,10);
                    resultSet.close();
                    query = "SELECT (b.activ_planned_finishdate + a.activ_duration) FROM
Activity a, Activity b WHERE a.activ_no = '" + inputActivityNo + "' AND b.activ_no =
'" + currentActivity + "' ;
                    resultSet = sqlStatement.executeQuery(query);
                    resultSet.next();
                    finishDateSQL = resultSet.getDate(1);
                    finishDate = resultSet.getString(1);
                    finishDate = finishDate.substring(0,10);
                    resultSet.close();
                    PreparedStatement preparedStatement =
jdbcConnection.prepareStatement("UPDATE Activity SET activ_predecessor = ? ,
activ_successor= ?, activ_planned_startdate = ? , activ_planned_finishdate = ? WHERE
activ_no = ?");
                    preparedStatement.setString(1, currentActivity);
                    preparedStatement.setString(2, successorActivity);
                    preparedStatement.setDate(3, startDateSQL);
                    preparedStatement.setDate(4, finishDateSQL);
```

```
                preparedStatement.setString(5, inputActivityNo);
                int rowsAffected = preparedStatement.executeUpdate();
                System.out.println(" The new start date is : " + startDate);
                System.out.println("The new finish date is : " +  finishDate);
                System.out.println(" Before activity : " + preAct);
                System.out.println(" After Activity : " +  sucAct);
                sqlStatement.executeUpdate("COMMIT");
                }
                currentActivity = successorActivity;
        }
        if (! slotFound){
                System.out.println("Sorry there is no room to reschedule this activity ");
        }
}
System.out.println("=======================================");
  }
}

public void shiftActivity(Statement sqlStatement,ResultSet resultSet,Connection
jdbcConnection)  throws Exception{
        String inputactivity = null; int dateDiff = 0; int shiftAmount = 0;
        int rowsAffected = 0; String query = null;
        System.out.println("=======================================");
        System.out.println("Which activity would you like to shift ?");
        try {
                InputStreamReader in1 = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(in1);
                inputactivity = br.readLine();
        }catch(Exception e){
                System.out.println(e.getMessage());
        }
        System.out.println("You have entered the activity "  + inputactivity);
        /****** GET THE DIFFERENCE OF TWO DATES **********/
        query = "SELECT (B.Activ_planned_startdate - A.activ_planned_finishdate)
FROM ACTIVITY A, ACTIVITY B WHERE A.activ_successor = B.activ_no AND
A.activ_no = '" + inputactivity + "'";
        resultSet = sqlStatement.executeQuery(query);
        resultSet.next();
        dateDiff = resultSet.getInt(1);
        resultSet.close();
        if (dateDiff > 0){
                System.out.println("You can shift this activity at most " + dateDiff + "
day(s) ");
                System.out.println("Please enter the amount of days to shift ? ");
                try {
                        InputStreamReader in1 = new InputStreamReader(System.in);
```

```
                        BufferedReader br = new BufferedReader(in1);
                        shiftAmount = Integer.parseInt(br.readLine());
                }catch(Exception e){
                        System.out.println(e.getMessage());
                }
                System.out.println("You have entered " + shiftAmount + " day(s) to shift
the activity " + inputactivity);
                PreparedStatement preparedStatement =
jdbcConnection.prepareStatement("UPDATE Activity SET activ_planned_startdate =
(activ_planned_startdate + ?), activ_planned_finishdate = (activ_planned_finishdate + ?)
WHERE activ_no = ?");
                preparedStatement.setInt(1, shiftAmount);
                preparedStatement.setInt(2, shiftAmount);
                preparedStatement.setString(3, inputactivity);
                rowsAffected = preparedStatement.executeUpdate();
                sqlStatement.executeUpdate("COMMIT" );
                System.out.println("Activity " + inputactivity + " shifted " + shiftAmount
+ " days ");
        }else{
                System.out.println("Sorry, there is no slot to shift the activity " +
inputactivity);
        }
        System.out.println("=======================================");
}

public static void changeActivity(Statement sqlStatement,ResultSet resultSet,Connection
jdbcConnection)  throws Exception
{
        String input_Activity = null;  String current_Activity = null;
        String successor_Activity = null;  boolean noRecord = false;
        String old_Start_Date = null;  String finish_Date = null;
        String start_Date = null; String query = null;
        java.sql.Date finish_DateSQL = null; java.sql.Date start_DateSQL = null;
        int dur = 0; int date_Difference = 0;
        String first_Link = null; String last_Link = null;
        System.out.println("=======================================");
        System.out.println("Select the activity that you would like to change its place");
        try {
                InputStreamReader in1 = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(in1);
                input_Activity = br.readLine();
        }catch(Exception e){
                System.out.println(e.getMessage());
        }
        if (input_Activity.equals("P001")){
                System.out.println("First Activity "+ input_Activity + " can not be
```

```
rescheduled ...");
        }else if (input_Activity.equals("R001")){
                  System.out.println("Last Activity "+ input_Activity + " can not be
rescheduled ...");
        }else{
                System.out.println("After which activity would you like to place the
activity " + input_Activity);
        try {
                InputStreamReader in1 = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(in1);
                current_Activity = br.readLine();
        }catch(Exception e){
                System.out.println(e.getMessage());
        }
        if (input_Activity.equals(current_Activity)){
                System.out.println("You can not choose the same activity");
          }else{
        /******* GET THE DURATION OF THE ACTIVITY SELECTED *******/
                query =  "SELECT activ_duration,activ_planned_startdate FROM
Activity WHERE Activ_no =" + "'" + input_Activity + "'";
                noRecord = true;
                resultSet = sqlStatement.executeQuery(query);
                while (resultSet.next()){
                        dur = resultSet.getInt(1);
                        old_Start_Date = resultSet.getString(2);
                        old_Start_Date = old_Start_Date.substring(0,10);
                        noRecord = false;
                }
                resultSet.close();
                if (noRecord){
                        System.out.println(input_Activity + " activity could not be found
");
                }else{
                        System.out.println("activity no: "  + input_Activity);
                        System.out.println("oldstartdate: " + old_Start_Date);
                        System.out.println("duration: " + dur);
                        /**GET THE DIFFERENCE OF TWO DATES ****/
                        query = "SELECT (B.Activ_planned_startdate -
A.activ_planned_finishdate) FROM ACTIVITY A, ACTIVITY B WHERE
A.activ_successor = B.activ_no AND A.activ_no = '" + current_Activity + "'";
                        resultSet = sqlStatement.executeQuery(query);
                        resultSet.next();
                        date_Difference = resultSet.getInt(1);
                        resultSet.close();
                        if(date_Difference >= dur){
                                query = "SELECT Activ_successor FROM Activity
```

```
WHERE activ_no=" + """+ current_Activity + """;
                                    resultSet = sqlStatement.executeQuery(query);
                                    resultSet.next();
                                    successor_Activity = resultSet.getString(1);
                                    resultSet.close();
                                    /* get firstlink and lastlink */
                                    query = "SELECT activ_predecessor, activ_successor
FROM Activity WHERE activ_no='" + input_Activity + "'";
                                    resultSet = sqlStatement.executeQuery(query);
                                    resultSet.next();
                                    first_Link = resultSet.getString(1);
                                    last_Link = resultSet.getString(2);
                                    resultSet.close();
                                    sqlStatement.executeUpdate("UPDATE Activity SET
activ_successor = '" + input_Activity + "' WHERE activ_no = '" + current_Activity + "'");
                                    sqlStatement.executeUpdate("UPDATE Activity SET
activ_predecessor = '" + input_Activity + "'   WHERE activ_no = '" + successor_Activity
+ "'");
                                    sqlStatement.executeUpdate("UPDATE Activity SET
activ_successor = '"  + last_Link + "' WHERE activ_no = '" + first_Link + "'");
                                    sqlStatement.executeUpdate("UPDATE Activity SET
activ_predecessor = '"  + first_Link + "' WHERE activ_no = '" + last_Link + "'");
                                    query = "SELECT (activ_planned_finishdate + 1) FROM
Activity WHERE activ_no = '" + current_Activity + "'";
                                    resultSet = sqlStatement.executeQuery(query);
                                    resultSet.next();
                                    start_DateSQL = resultSet.getDate(1);
                                    start_Date = resultSet.getString(1);
                                    start_Date =start_Date.substring(0,10);
                                    resultSet.close();
                                    query = "SELECT (b.activ_planned_finishdate +
a.activ_duration) FROM Activity a, Activity b WHERE a.activ_no = '" + input_Activity
+ "' AND b.activ_no = '" + current_Activity + "' ";
                                    resultSet = sqlStatement.executeQuery(query);
                                    resultSet.next();
                                    finish_DateSQL = resultSet.getDate(1);
                                    finish_Date = resultSet.getString(1);
                                    finish_Date =finish_Date.substring(0,10);
                                    resultSet.close();
                                    PreparedStatement preparedStatement =
jdbcConnection.prepareStatement("UPDATE Activity SET activ_predecessor = ? ,
activ_successor= ?, activ_planned_startdate = ? , activ_planned_finishdate = ? WHERE
activ_no = ?");
                                    preparedStatement.setString(1, current_Activity);
                                    preparedStatement.setString(2, successor_Activity);
                                    preparedStatement.setDate(3, start_DateSQL);
```

```
                                    preparedStatement.setDate(4, finish_DateSQL);
                                    preparedStatement.setString(5, input_Activity);
                                    int rowsAffected = preparedStatement.executeUpdate();
                                    System.out.println(" The new start date is : " + start_Date);
                                    System.out.println("The new finish date is : " +
finish_Date);
                                    sqlStatement.executeUpdate("COMMIT" );
                            }else{
                                    System.out.println("Sorry, there is no enough slot to place
the activity " +  input_Activity + " after the activity " + current_Activity);
                            }
                    }
            }
            System.out.println("=======================================");
    }
}
}
```

LIST OF REFERENCES

1. Hammer J, Schmalz M, O'Brien W, Shekar S, Haldavnekar N. SEEKing knowledge in legacy information systems to support interoperability. CISE Technical Report. Gainesville: University of Florida; 2002b August. Report No.: CISE TR02-008.

2. Sangeetha S, Hammer J, Schmalz M, Topsakal O. Extracting Meaning from Legacy Code through Pattern Matching, Technical Report TR-03-003, July 2003

3. Hammer J, Schmalz M, O'Brien W, Shekar S, Haldavnekar N. SEEKing knowledge in legacy information systems to support interoperability. In ECAI 2002. Proceedings of International Workshop on Ontologies and Semantic Interoperability; 2002 July 23; Lyon, France; 2002c. p. 67-74.

4. Kouznetsov P. "JAD - The Fast Java Decompiler", http://kpdus.tripod.com/jad.html, Accessed July 29, 2003.

5. The Source for Java(tm) Technology Collaboration, "Java Compiler Compiler (JavaCC)", http://javacc.dev.java.net/, Accessed July 29, 2003.

6. The Cooperative Database Project Public Service Web Page at UCLA, "JavaCC Grammar Repository", http://www.cobase.cs.ucla.edu/pub/javacc/, Accessed July 29, 2003.

7. Java Tree Builder Homepage at Purdeu University, "Java Tree Builder (JTB)", http://www.cs.purdue.edu/jtb/, Accessed July 29, 2003.

8. Palsberg J, Jay B. The Essence of the Visitor Pattern. In 22nd International Computer Software and Applications Conference (COMPSAC), 1998 August 19-21; Vienna, Austria.

9. Bravenboer M, Visser E. Guiding visitors: Separating navigation from computation. Technical Report UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University, 2001

10. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Pub Co. ISBN 0-201-63361-2.

11. Backus J. The syntax and semantics of the proposed international algebraic language of information processing. International Conference on Information Processing, Paris, France, 1959.

12. Hecht M, Flow Analysis of Computer Programs. Amsterdam, Holland: Elsevier North- Holland Publishing Co., 1977.

13. Weiser M. Program slicing. In ICSE 1981. Proceedings of the 5th International Conference on Software Engineering; 1981 March 9-12; San Diego, California; 1981. p. 439-449.

14. Shao J, Pound C. Extracting Business Rules From Information Systems, BT Technology Journal vol 17 no 4 (1999) pp. 179-186 ISSN 1358-3948

15. Hay D, Healy K A. Defining Business Rules. Final Report in the GUIDE Business Rule Project. http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf, Accessed 29 July 2003.

BIOGRAPHICAL SKETCH

Oguzhan Topsakal was born on December 4, 1974 in Malatya, Turkey. He received his Bachelor of Science degree from the Computer and Control Engineering Department of Istanbul Technical University in June 1996. He worked for several companies where he had the opportunity to apply his theoretical knowledge to real-life problems.

He joined the department of Computer and Information Science and Engineering in August 2001. He worked as a teaching assistant for several courses such as Data Structures and Algorithms, Discrete Mathematics, and Introduction to Computer Science. He also worked as a teaching assistant for the Data Warehousing, Decision Support and Data Mining course at the University of Honk Kong. He also worked as a research assistant under Dr. Joachim Hammer in Scalable Extraction of Enterprise Knowledge (SEEK) project. He completed his Master of Science degree in computer engineering at the University of Florida, in August 2003. His research interests include algorithms, data mining and data warehousing.