2007

Reverse Engineering Software Code in Java to Show Method Level Dependencies

Lesley B. Hays
University of North Florida

Suggested Citation

Hays, Lesley B., "Reverse Engineering Software Code in Java to Show Method Level Dependencies" (2007). *UNF Graduate Theses and Dissertations*. 193.
https://digitalcommons.unf.edu/etd/193

REVERSE ENGINEERING SOFTWARE CODE IN JAVA TO SHOW METHOD LEVEL DEPENDENCIES

by

Lesley B. Hays

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December 2007

The thesis "Reverse Engineering Software Code in Java to Show Method Level Dependencies" submitted by Lesley Hays in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature deleted

12/9/2007

Dr. Robert Roggio
Thesis Advisor and Committee Chairperson

Signature deleted

12/9/2007

Dr. Neal Coulter

Signature deleted

12,9,07

Dr. Behrooz Seyed-Abbassi

Accepted for the School of Computing:

Signature deleted

12/11/07

Dr. Judith Solano
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature deleted

12/9/2007

Dr. Neal Coulter
Dean of the College

Accepted for the University:

Signature deleted

12 DECEMBER 2007

Dr. David W. Fenner
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to thank my family for their continuous support, encouragement, and understanding during these past few years, thank you! I would like to express my sincerest gratitude to all my friends who have offered their assistance to me. They have been the greatest resource anyone could ask for. Thank you! I would also like to thank the faculty and staff at UNF, including Dr. Coulter and Dr. Abbassi, for always guiding me and for always expecting my best. Finally, I would like to thank Dr. Roggio for his help and guidance throughout, not only my thesis work, but my many years at UNF. I truly appreciate all your time and assistance and feel I have learned a lot from this experience. Thank you!

TABLE OF CONTENTS

FIGURES

ABSTRACT

With the increased dependency on the Internet and computers, the software industry continues to grow. However, just as new software is being developed, older software is still in existence and must be maintained. This tends to be a difficult task, as the developers charged with maintaining the software are not always the developers who designed it. Reverse engineering is the study of an application's code and behavior, in order to better understand the system and its design. There are many existing tools that will assist the developer with this undertaking, such as Rational Rose®, jGRASP®, and Eclipse®. However, all the tools generate high level abstractions of the system in question, like the class diagram. It would be more beneficial to developers to have illustrations with more detailed information, such as the method level dependencies in the source code. In order to accomplish this task, a new framework has been developed that will allow the user to view both high level and lower level code detail. As users attempt to perform code maintenance, they will run the code through an existing tool, such as Rational Rose®, and then through the Method Level Dependency Generator component, to show the method level dependencies. These tools used together provide the software maintainer with more useful information, assisting with the software development process, including code design, implementation, and testing.

Chapter 1

INTRODUCTION

In the world of computing applications, approximately 30-35% of the overall total life-cycle costs are devoted to helping the programmer understand the functionality of existing code. This is a necessary task, in order to correctly make required changes in response to new requirements, to resolve errors, or perform other changes [Tomic94]. A thorough understanding of the logic, design, and structure of existing code will help developers, management, and analysts more accurately estimate the maintenance and enhancement costs, analyze code complexity, undertake thorough testing, and estimate software reliability more effectively and efficiently. However, with the "time is money" mentality that dominates in most workplaces, a professional is rarely given a sufficient amount of time to thoroughly and comprehensively complete a task in a manner that does not introduce additional problems in the software.

Reverse engineering, "… [analyzes a] system's code, documentation and behavior to create system abstractions and design information" [Ali05]. Reverse Engineering is, essentially, the practice of examining existing systems, at any stage, to identify elements and dependencies. This information is then used to gain more knowledge about the design, the structure, system code, and functionality.

There are many existing tools, such as Rational Rose®, jGRASP®, NetBeans®, and Eclipse® (to name but a few), that provide a degree of reverse engineering. Several tools and frameworks take Java code as input and generate the Unified Modeling Language (UML) class diagrams. These diagrams are helpful to the users by illustrating object dependencies; however, they tend to be high level and leave much to be desired about "lower level" (i.e., code level) application specifics. While object dependencies are indicated through UML associations, multiplicity, direction, and other real-world objects can be complex. General dependencies at this architectural level (class diagrams with dependencies) are helpful, but such renderings leave the professional in dire need of much more detailed analysis of object dependencies extending down to method-level dependencies, which is where actual code maintenance will occur.

As a developer, it would be more beneficial to have a framework that drills down a level further than providing high-level class dependencies. A comprehensive reverse engineering framework that, when given an unknown Java program, will analyze the existing structural characteristics and generate detailed low-level dependencies and relationships among code segments would be helpful in a workplace environment. Such a framework would be used in conjunction with a well documented tool, such as Rational Rose®, that already generates the UML class diagram to form a more comprehensive maintenance approach. These existing tools would be used to show the basic architectural relationships followed by this new framework that focuses on detailed relationships by providing two-fold forward and reverse analyses of method level dependencies, offering a more practical tool for software maintenance. The framework would, by class, show all methods declared in the class and what methods they invoke. It would also, by each method, show the class and methods it is referenced by. Equivalently, "who" invokes the services of this class and what services of other classes does "this" class invoke would be shown.

While this is clearly an arduous undertaking, a framework that provides this level of analysis up front to a software professional before starting a software maintenance task has multiple benefits. It should assist the user in both understanding of the design and complexity of an existing application as well as assuring the user a more reliable maintenance undertaking.

To set the stage for this undertaking, this research first presents a number of popular development frameworks containing reverse engineering tools, such as Rational Rose®, jGRASP®, NetBeans®, Eclipse® and others, in order to comprehensively identify both their strengths and shortcomings. The thesis will then present the details of the new framework that provides detailed method dependencies and associations.

Chapter 2

REVIEW OF THE LITERATURE

2.1 Reverse Engineering

2.1.1 Related Areas and Sub-Topics in Reverse Engineering

Reverse engineering is a broad subject area, which includes a variety of sub-topics and components. Many terms are used when discussing reverse engineering. Some of these terms include [Tomic94]:

- Forward Engineering – the process of starting at the gathering of requirements and then following through to design and finally to the implementation of the application.
- Design Recovery – gathering additional information, like domain knowledge, outside information, and deductive information for inclusion with other observations, to assist the professional in better understanding the system being studied.
- Restructuring – the movement from one form to another form at the same level of abstraction without changing the system's output. Essentially, it is changing code to put it in a more structured format.

- Reengineering – the investigation and modification of a system to rebuild it in a new form. It is usually accomplished by reverse engineering a system and then forward engineering the system.
- Software Maintenance - includes changing source code to correct errors, improve performance, fix problems, etc.

2.1.2 Reverse Engineering Defined

With society's dependence on the Internet, many businesses need to modify their current applications, to make them web-based and move towards an electronic way of doing business. This trend has created more of an interest in code maintenance and evolution than in the past [Ali05]. Thus, there is now a need for experts in older systems, as software maintenance and evolution is becoming more necessary. Roughly, one third of total life-cycle costs are used for the programmer to understand the functionality of the existing code [Tomic94]. Even though it is a timely and costly process, understanding the code is critical and significant, in order for a programmer to correctly make the desired changes. Software maintenance and evolution continue to become more important as time marches on.

Reverse engineering is the act of recognizing systems elements, along with their corresponding dependencies, to generate a variety of application abstractions and design data from these system elements [Muller00]. To successfully do this with software, the application's code, documentation, and behavior must be studied to identify the system abstractions and various design patterns, as well as to fully understand the functionality of the system.

Software reverse engineering may be viewed as a "solution looking for a problem." [Buss91]. Many programmers attempt to understand "how the code gets where it's going" and "why the code is doing something" in their everyday tasks. While there are many different approaches and techniques to reverse engineer software, their common goal is to gather as much information as possible from the current system, to assist in the maintenance task(s) at hand. This information is critical to support current maintenance and/or future development, as well as providing data to project management for planning the use of software engineering resources.

2.1.3 History of Reverse Engineering

The need for reengineering legacy systems was apparent by the early 1990s. However, with the recent pressure for businesses to go electronic, by way of the Internet, to and convert many existing systems to web-based applications, this need has intensified. There is now a demand for various methods, tools, and infrastructures to assist in transforming existing applications rather quickly and relatively inexpensively [Muller00]. Over the past decade, researchers have made tremendous advances in this area.

The 1980s were focused on various program comprehension theories, along with identifying the concept of reverse engineering with the evolution of software. It was noted that a majority of the software evolution process is used up by program comprehension. The topic continued to be researched throughout the 1990s. It was during this time that various infrastructures and tools were developed to assist with the main parts of reverse engineering a system [Muller00]. As long as an application is used, it will continuously change. As it changes, it will become more and more complex.

2.1.4 Problems with Reverse Engineering

Software reverse engineering is difficult for many reasons. One reason is there might not be any documentation in the code to be modified. In some instances, the code may be complex, making the original author's purpose difficult for the new engineer to understand. Another issue occurs when the original code does not provide the correct solution for the problem. The code may have also been altered from additional problems found, creating a very cluttered and disorganized environment with which to work. The programming language may have been updated, causing new problems in the code. The software could have come from a different environment or the hardware platform may have been modified. These are just a few of the problems software engineers may face while trying to maintain code [Buss91]. If software engineers do not fully understand the code they are modifying, this can create future problems.

2.1.5 Importance of Reverse Engineering

Approximately $30 billion is spent a year on software maintenance, including legacy systems [Tomic94]. An important and poor trait of legacy systems is many times business rules are intertwined within the application logic. As software lives, it is updated due to enhancements in the functionality, correction of errors, or improvements in quality. However, as software changes, the documentation is not always updated, as well. Therefore, the code becomes the only dependable source of information when trying to understand the application's functionality. Previous design, if available, does not always map to the current implementation. Yet, effective maintenance requires a reasonably thorough understanding of the code and its intended functionality. This has led to the need for reverse engineering or some mechanism to recapture some of the original design intentions. By reverse engineering an application's code a user can then recognize the artifacts, detect various relationships, and produce abstractions that can be used to re-document and depict the initial design.

2.1.6 Practicality of Reverse Engineering

When maintaining old code, the organization will eventually need to decide if it is most cost effective to keep maintaining the existing code or if the organization should reengineer the system. There are many factors used when determining if system reengineering is appropriate. A system should be reengineered if there are regular failures, code that is out of date (about seven-to-ten years old), using application logic or structure that is excessively complicated, or code written for hardware that is obsolete [Buss91]. Other factors to consider for reengineering are when there is code with exceedingly large modules, unnecessary resource usage, aspects in the code that are hard-coded, difficulty in keeping resources to maintain the code, documentation that is lacking and leaves much to be desired, or unfinished design specifications [Buss91]. By reengineering a system, the maintainability will be improved, migration to a new environment is easier, the system tends to be more reliable, and the code is more prepared for functional enhancements.

Another reason to want to have a thorough understanding of code is the size of many applications. As they increase in size and become more complex, it becomes more important to understand their structure and behavior. Reverse engineering the code will help bring that knowledge to the user. Often, there is little information or rationale documented behind the implementation decisions. Reverse engineering is, therefore, sometimes vital to understand the reason and logic behind existing code.

2.2 Reverse Engineering Tools

Most reverse engineering tools available, including Rational Rose®, NetBeans®, and Eclipse®, will generate a UML class diagram from Java source code. jGRASP® goes somewhat deeper by generating the class diagram, a Control Structure Diagram (CSD) which is an algorithmic level diagram, and a Viewer which will display dynamic visualizations of objects and primitives. Eclipse® provides for some additional reverse engineering functionality within the environment itself.

The tools mentioned are the more popular reverse engineering tools in common use. However, there are many others tools, including the Sun Java Studio Enterprise 7® or JBuilder®, to name a few, that will perform various software reverse engineering functions, as well. These tools will execute a variety of tasks, in addition to some of the same operations as the other tools. However, all the tools excel in different ways and possess different levels of capabilities, some are just more widely used then others.

2.2.1 Rational Rose

Rational Rose® is a modeling tool, released by the Rational Software Corporation (recently purchased by IBM), that supports, among a host of additional features, the UML graphical notation [IBM07]. An example of a class diagram is shown in Figure 1. Rational Rose® will automatically generate a UML class diagram from object oriented source code, such as Java and C++. This is a good tool for round trip engineering, as it will allow you to create UML class diagrams from existing code, modify them, and update the source code immediately inside the application. However, there is still a good deal of human interaction required during this process. While this approach is helpful, there is still a lot to be desired when trying to understand method dependencies, necessary, for complete programmer comprehension of the system workflow and logic essential in application maintenance.

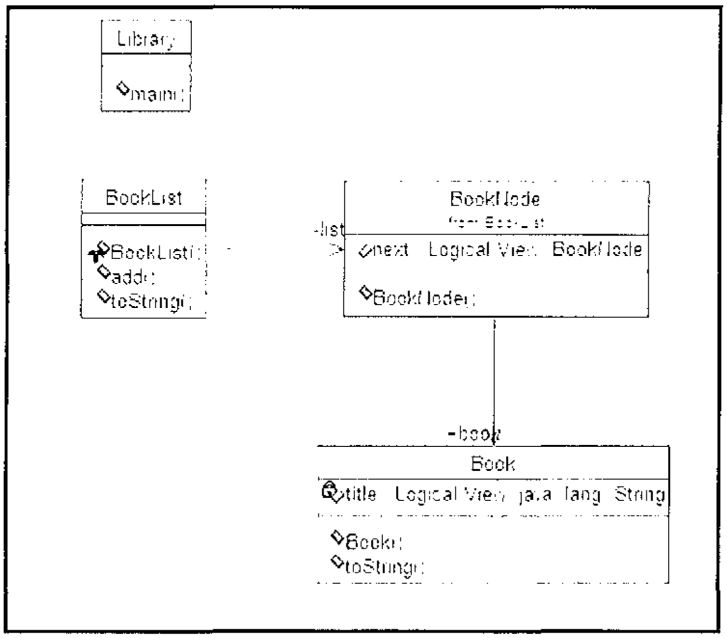
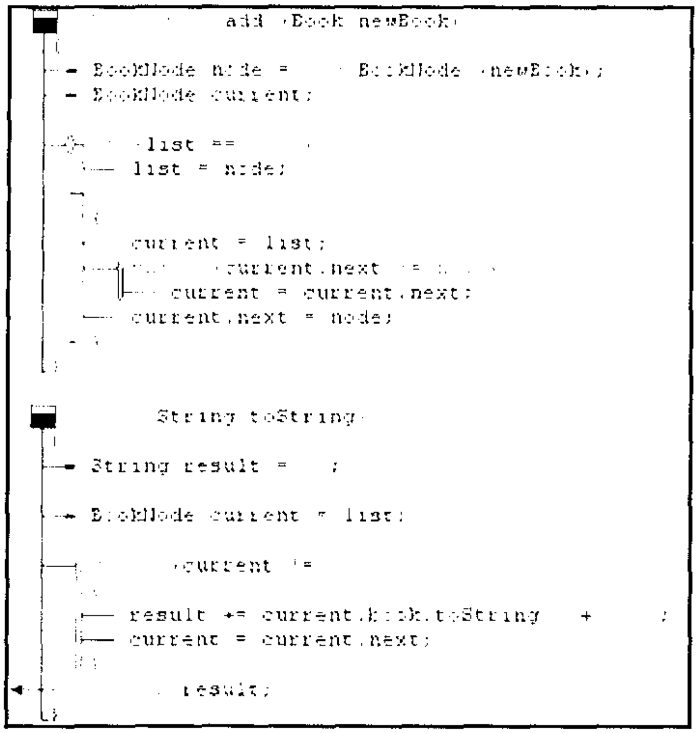


Figure 1: Rational Rose UML Class Diagram

2.2.2 jGRASP

jGRASP® is developed from pcGRASP. jGRASP® is one of the most recent applications from the GRASP (Graphical Representations of Algorithms, Structures, and Processes) group at Auburn University [Auburn University07]. The application jGRASP® is a "lightweight integrated development environment, created specifically to provide visualizations for improving the comprehensibility of software" [Auburn University07]. jGRASP® is written in Java and supports the Java programming language, as well as C, C++, and Ada. jGRASP's current functionality includes the automatic generation of CSDs, UML class diagrams, and Viewers. jGRASP® also contains an Object Workbench and Debugger, which help a programmer to generate and debug source code.

The CSD is an "algorithmic level diagram generated for Ada, C, C++, and VHDL" [1]. An example is shown in Figure 2. This diagram assists the user in understanding the source code more thoroughly and in an easier manner. It will do this by representing control constructs, control paths, and the general structure of each program segment. This diagram is illustrated in the margins and indentations of the source code. This diagram is often used in the place of flow charts and other graphical diagrams. The main purpose of the CSD diagram is to "create an intuitive and compact graphical notation that is easy to use" [Auburn University07].



```
          add (Book newBook)
{
   BookNode node =     BookNode (newBook);
   BookNode current;

      (list ==
      list = node;

   {
      current = list;
            (current.next !=
         current = current.next;
      current.next = node;
   }
}

       String toString
{
   String result =    ;

   BookNode current = list;

         (current !=

      result += current.book.toString   +      ;
      current = current.next;

      result;
}
```

Figure 2: jGRASP CSD Diagram

jGRASP® will also generate the UML class diagram, as shown in Figure 3, for the Java source code from the Java class files and .jar files of a project. The diagram will illustrate the dependencies among various classes by standard UML dependency arrows. If the user selects a class, its members are displayed. If the user selects an arrow, the dependencies between the two classes are illustrated. This diagram will help the user comprehend the high-level elements and dependencies among the classes for the specified program.

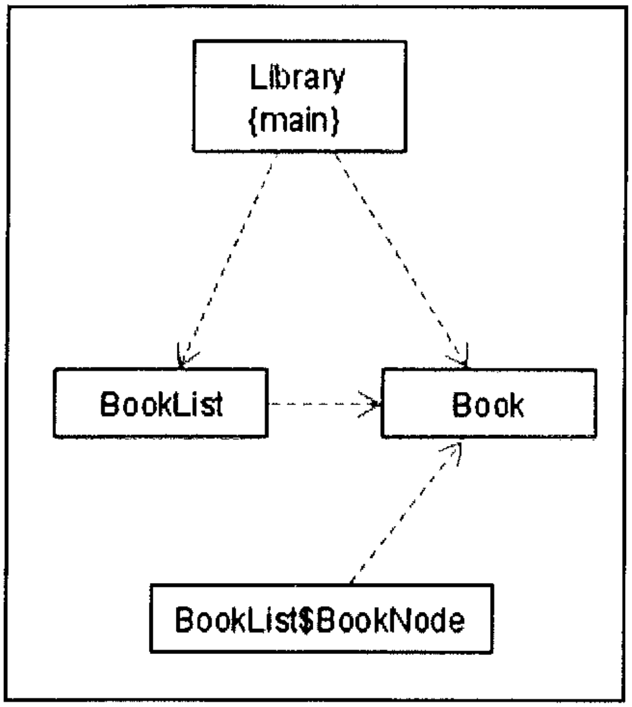
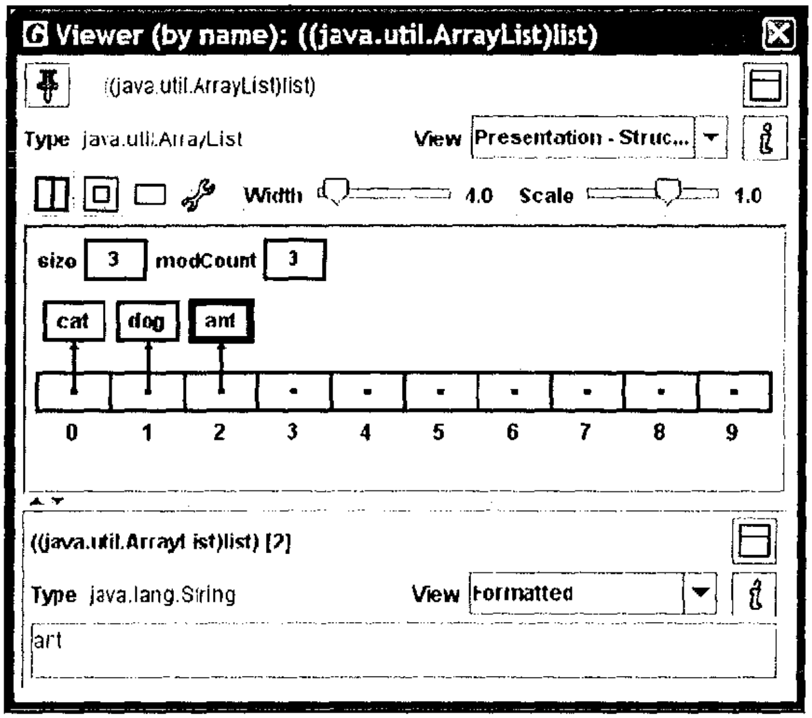


Figure 3: jGRASP UML Class Diagram

jGRASP® will also generate Viewers for Java source code, as illustrated in Figure 4. The Viewers, "for objects and primitives provide dynamic visualizations as the user steps through a program in debug mode or invokes a method for an object on the workbench" [Tilley01]. Presentation views are presented for instances of classes that symbolize data structures, such as a link list, binary trees, and array wrappers. When the user opens a viewer, a structure identifier recognizes the data structure during the debugging process and displays the correct presentation view of the object for the user.



Viewer (by name): ((java.util.ArrayList)list)

((java.util.ArrayList)list)

Type java.util.ArrayList View Presentation - Struc...

Width 1.0 Scale 1.0

size 3 modCount 3

((java.util.ArrayList)list) [2]

Type java.lang.String View Formatted

art

Figure 4: jGRASP Viewer Diagram
[Auburn University07]

jGRASP® is a very useful tool in helping the user understand existing code by generating the CSD diagrams. However, it was noted that the UML documentation generation feature is not as complete, therefore, not as supportive as it could be to the developer. This tool is very useful when attempting to debug and understand code. However, there are still some important features that could be added to assist the user fully.

2.2.3 NetBeans

The NetBeans® Integrated Development Environment (IDE) is an open source application for the development and maintenance of Java application code [NetBeans 07]. NetBeans® will create an UML class diagram from object-oriented source code, such as Java and C++ (Figure 5). This tool will allow a software engineer to create UML class diagrams from existing code. The class diagram will allow the user to see potential object dependencies, thus, helping the user understand the code. However, high level, graphical object dependencies only provide limited insight to the developer. More information is vital to foster a firm grasp of what exactly is going on throughout the application logic.

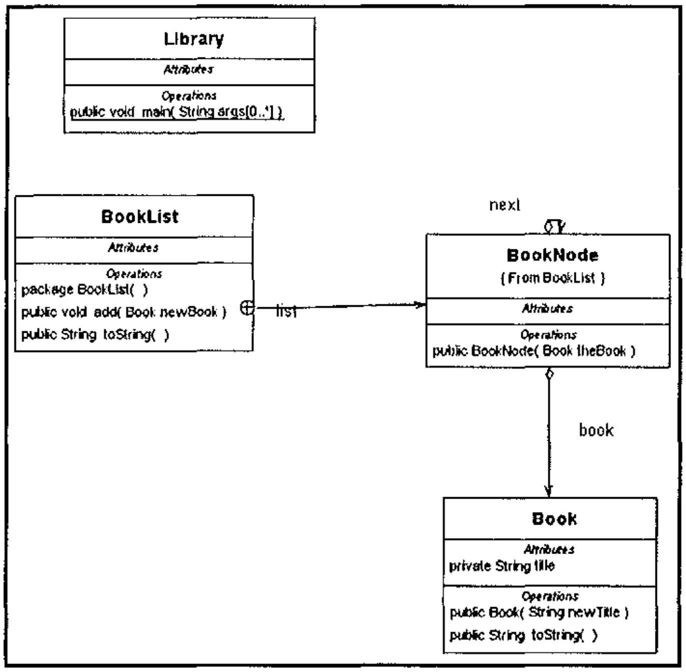


Figure 5: NetBeans UML Class Diagram

2.2.4 Eclipse

Eclipse®, another product of IBM, is an open source tool that provides an advanced development environment for various applications [Eclipse07]. Eclipse® will allow a software engineer various reverse engineering techniques while in the Eclipse® workspace. The Smart Development Environment (SDE) plugin for Eclipse® provides reverse engineering of Java code into UML class diagrams and output in a PDF or HTML format, entirely within the Eclipse® environment. Figure 6 displays an example UML class diagram generated in Eclipse®. In addition to these facilities, Eclipse® also provides for various functionalities within the workspace to assist in understanding program code. The Eclipse® Java IDE may assist the user by providing search capabilities for finding referenced code declarations and usages. It provides various tools for this purpose, including Open Declaration, References, Declarations, etc. The Open Declaration operation will open a class to the selected method. The References tool will show all the references in the project for that specific method. The Declarations utility will show the class in which that denoted method is declared. These features may be very helpful, but it is necessary for the user to be within the project; that is, looking at the source code. There is not a way to find method dependencies up front or without being "inside" the program code.

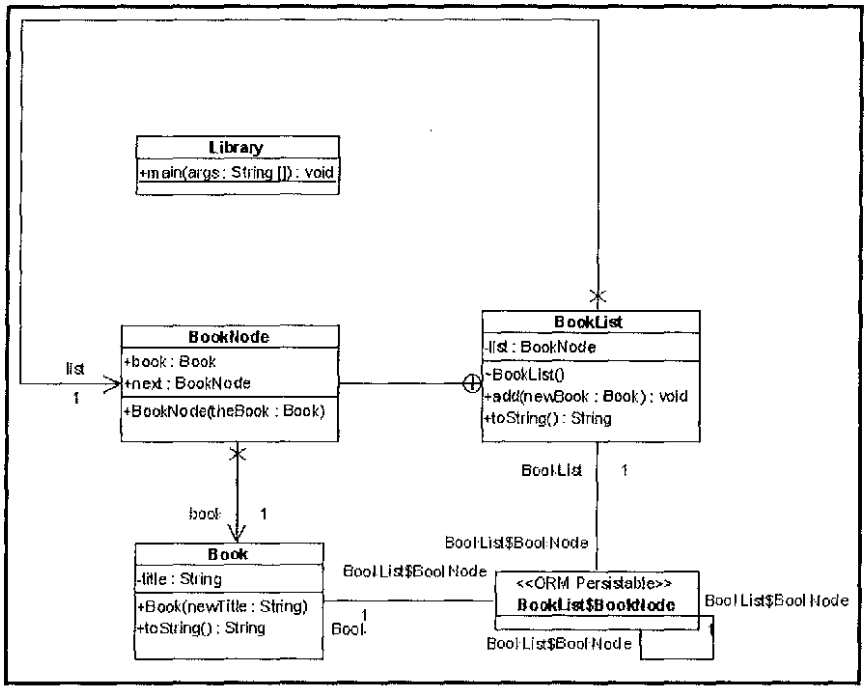


Figure 6: Eclipse UML Class Diagram

Chapter 3

METHODOLOGY

3.1 Method Level Dependency Framework

Research for this thesis included examining various reverse engineering tools, such as those found in Rational Rose®, Eclipse®, NetBeans®, and jGRASP®, followed by comparing and analyzing the their outputs and methodologies. Once these tools were evaluated, a new framework was developed that, when used in combination with an existing tool, will generate the UML class diagram, which is more beneficial during reverse engineering due to its focus on method level detail. This new reverse engineering framework included accepting Java programs as input and determining the structural characteristics of the program. It provides for both a forward and reverse analyses of method level dependencies. The framework provides two output diagrams: a complete listing by method of all classes and methods that reference the method in question, as well as an additional listing of all references made by each method in each class. While this is viewed by many as an arduous undertaking, the availability of such a framework, when used along with existing reverse engineering tools, should be helpful to the software maintenance worker. Figure 7 shows how the new method level dependency component fits into existing functionality, to assist the developer with software comprehension, thus, creating a new framework.

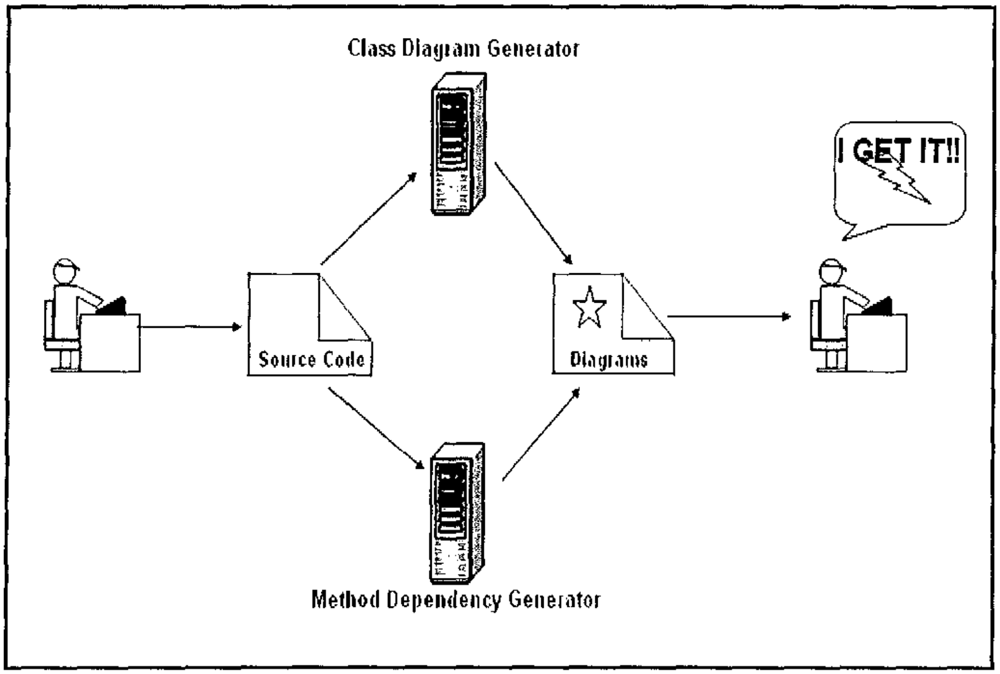


Figure 7: Method Level Dependency Framework

3.2 Reverse Engineering Framework

3.2.1 Development Software

Rational Rose®, jGRASP®, NetBeans®, and Eclipse® were used to generate the various models to support reverse engineering methodologies. The method level dependency framework was developed in Java 5.0 using the Eclipse® IDE. A MySQL® database was used for storage and retrieval of various information artifacts as needed. A machine containing the Java Run-Time Environment (JRE) was utilized to run the application. This is a stand alone application and runs locally on a machine.

3.2.2 Framework Development

3.2.2.1 Framework Design

The Method Level Dependency Generator component was developed using the Java programming language and was organized in a modular format. It consisted of five classes: MainFrame.java, FileHandler.java, DatabaseMethods.java, GenerateDiagrams.java, and Constants.java. Each class, composed of various methods, was designed to handle a different part of the application functionality. Figure 8 illustrates the class diagram for the Method Level Dependency Generator. From here, the various class dependencies can be seen, along with the global variables and methods found in each class.

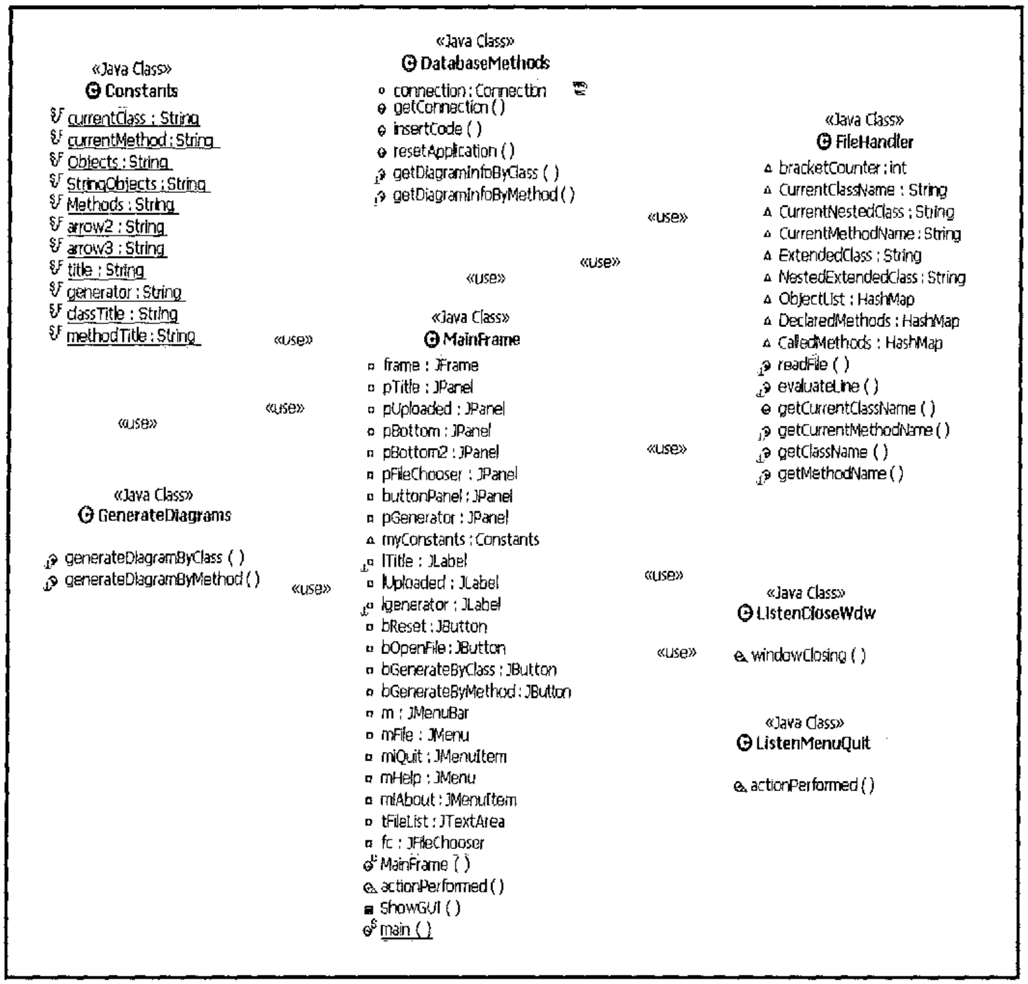


Figure 8: Class Diagram

3.2.2.1.1 MainFrame.java

MainFrame.java was designed to generate and handle the Graphical User Interface that runs as a stand alone application for this framework. The GUI was developed using the Swing toolkit in Java, which is part of the Java Foundation Classes. This toolkit allowed for easy use of standard components, such as textboxes, panels, buttons, frames, etc. This class contained the main() method . It built the GUI and controlled any action taken within the GUI by calling the corresponding methods to accomplish that task. The methods found in Mainframe.java are listed in Figure 9.

MainFrame.MainFrame()
MainFrame.actionPerformed()
MainFrame.ShowGUI()
MainFrame.main()
ListenMenuQuit.actionPerformed()
ListenCloseWdw.windowClosing()

Figure 9: MainFrame.java Method List

3.2.2.1.2 FileHandler.java

FileHandler.java managed and evaluated the data coming in through the input files. This class contained methods that read in the Java source code and, considering the order, examined it for the structural characteristics that would indicate a class or a method. First the entire input was read and any leading or trailing spaces and line feeds were removed, storing the input as a StringBuffer. Next, the input StringBuffer was parsed by open brackets, close brackets, or semi-colons, until the entire file was read. Each substring was evaluated to determine if it contained a class declaration, an object instantiation, a method declaration, or a method call. This was accomplished in Java through the use of regular expressions, also known as patterns. Regular

Expressions were created to recognize the class declarations, method declarations, all class instantiations, and any method calls in each given file.

- Class declarations were recognized by the keyword "class" and a space.
- Class instantiations were distinguished by the keyword "new" followed by a space, a word (characters a-z, A-Z, 0-9), and then a left parenthesis. The referenced name and the class name are both stored in a hash map for matching later.
- Method declarations were identified by a word (characters a-z, A-Z, 0-9) followed by a space, a word (characters a-z, A-Z, 0-9), and ending with a left parenthesis.
- The method calls were discovered by checking for a word (characters a-z, A-Z, 0-9) followed by a left parenthesis and checking to see if they have not already been labeled a class instantiation.

If part of the string matched a pattern, it was then checked for reserve words and parsed out by the characters, to get the actual class or method name. If a method call was found, it was stored in the database. The methods created to manipulate the input files are listed in Figure 10.

FileHandler.readFile()
FileHandler.evaluateLine()
FileHandler.getCurrentClassName()
FileHandler.getCurrentMethodName()
FileHandler.getClassName()
FileHandler.getMethodName()

Figure 10: FileHandler.java Method List

3.2.2.1.3 DatabaseMethods.java

DatabaseMethods.java manipulated the database. This class made the connection to the MySQL® database. It was also responsible for any calls to update or query the database throughout the framework. As method calls were identified by the FileHandler class, they were saved to the database. As the user selected to generate diagrams from the MainFrame class, this information was retrieved from the database. Figure 11 contains all the methods found in DatabaseMethods.java

DatabaseMethods.getConnection()
DatabaseMethods.insertCode()
DatabaseMethods.resetApplication()
DatabaseMethods.getDiagramInfoByClass()
DatabaseMethods.getDiagramInfoByMethod()

Figure 11: DatabaseMethods.java Method List

3.2.2.1.4 GenerateDiagrams.java

GenerateDiagrams.java was used to generate the output diagrams. The user has the ability to generate two different diagrams. The first diagram, “Generate by Class Dependencies,” will query the database and display, by class, each method and what method calls it makes. The second diagram, “Generate by Method Dependencies,” will query the database and illustrate by each method, what methods it is called by. The methods located in GenerateDiagrams.java are listed in Figure 12.

GenerateDiagrams.generateDiagramByClass()
GenerateDiagrams.generateDiagramByMethod()

Figure 12: GenerateDiagrams.java Method List

3.2.2.1.5 Constants.java

Constants.java defined all constants used throughout the framework. The Method Level Dependency Generator used constants to define the various patterns it was searching for, in each class and image locations. The constants used in the framework are provided in Figure 13.

Constants.currentClass
Constants.currentMethod
Constants.Objects
Constants.StringObjects
Constants.Methods
Constants.arrow2
Constants.arrow3
Constants.title
Constants.generator
Constants.classTitle
Constants.methodTitle

Figure 13: Constants.java Constants List

3.2.2.2 Database Design

The MySQL® database created to store the information was named "thesis." The thesis database only contained one table, "code." This table consisted of four columns, CurrentClass, CurrentMethod, CalledClass, and CalledMethod. While scanning the input file, as a method call was found in a class, the current class, current method, called class, and called method were stored in the code table. This table was queried, in order to generate the diagrams. The database diagram is found in Figure 14.

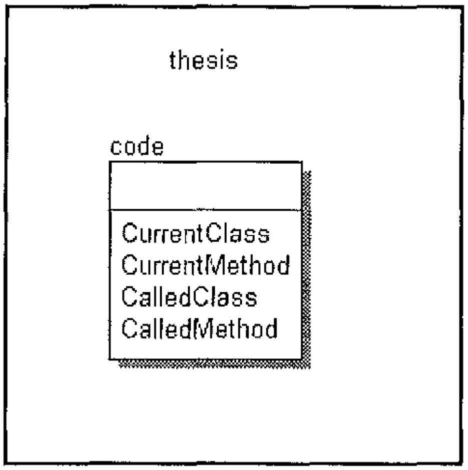


Figure 14: Database Diagram

3.2.3 Framework Functionality

The reverse engineering framework was relatively simple to operate and assumed the input java files would compile together. The user began by starting the application.

They were given a graphical user interface that would allow them to manipulate the framework. This GUI is shown in Figure 15.

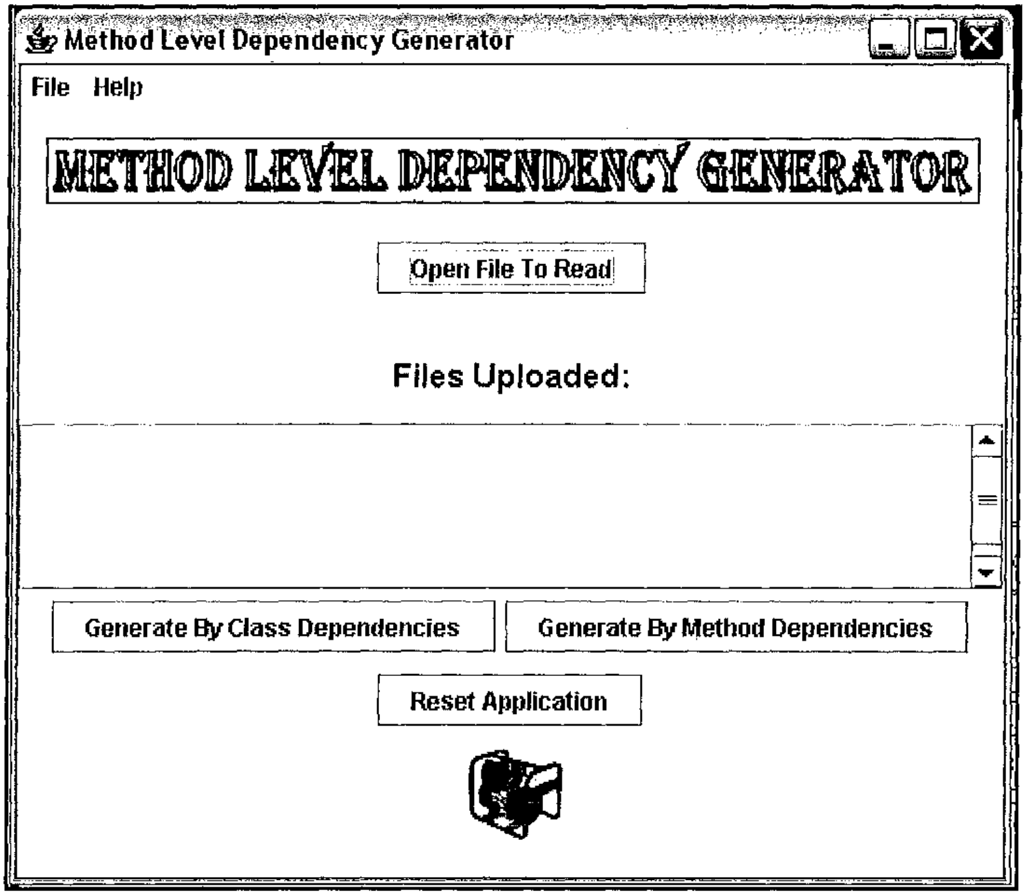
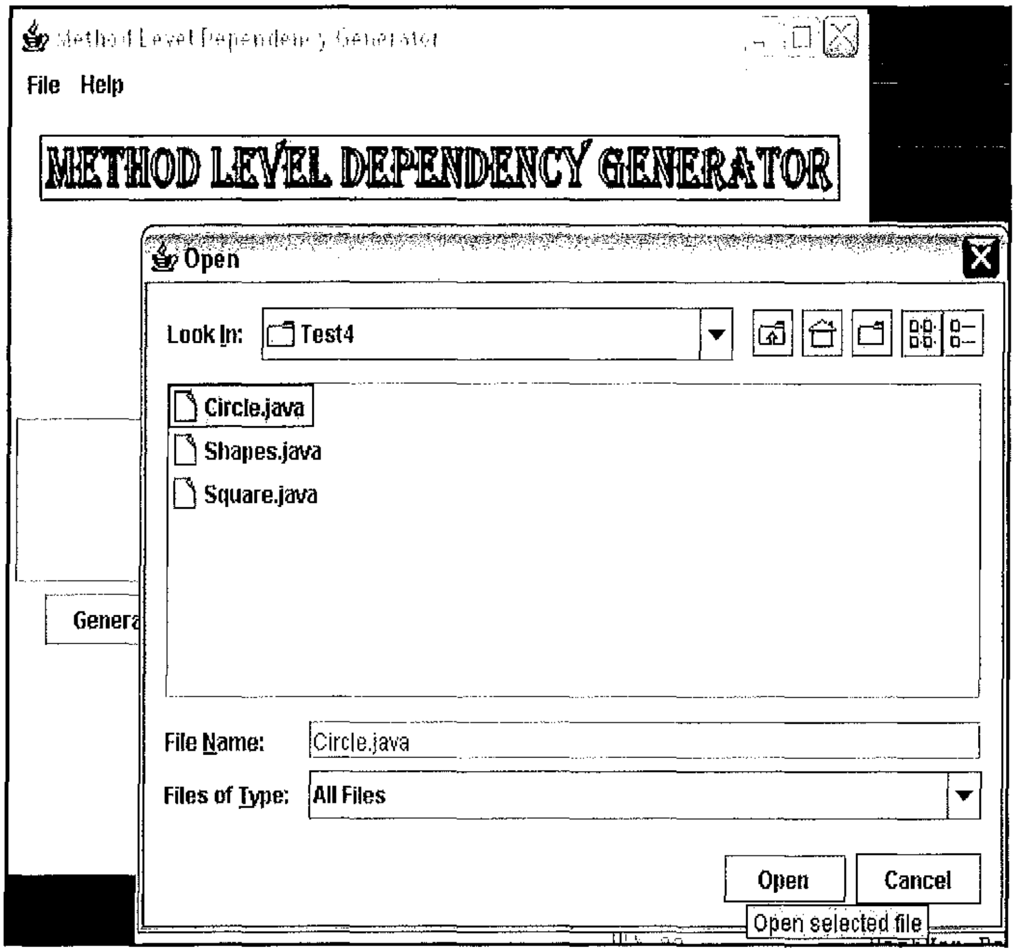


Figure 15: Method Level Dependency Generator

The user selected one file at a time to examine, by selecting the "Open File to Read" button and choosing the file. A file selector would appear and the user had to browse to find the desired file, as shown in Figure 16.



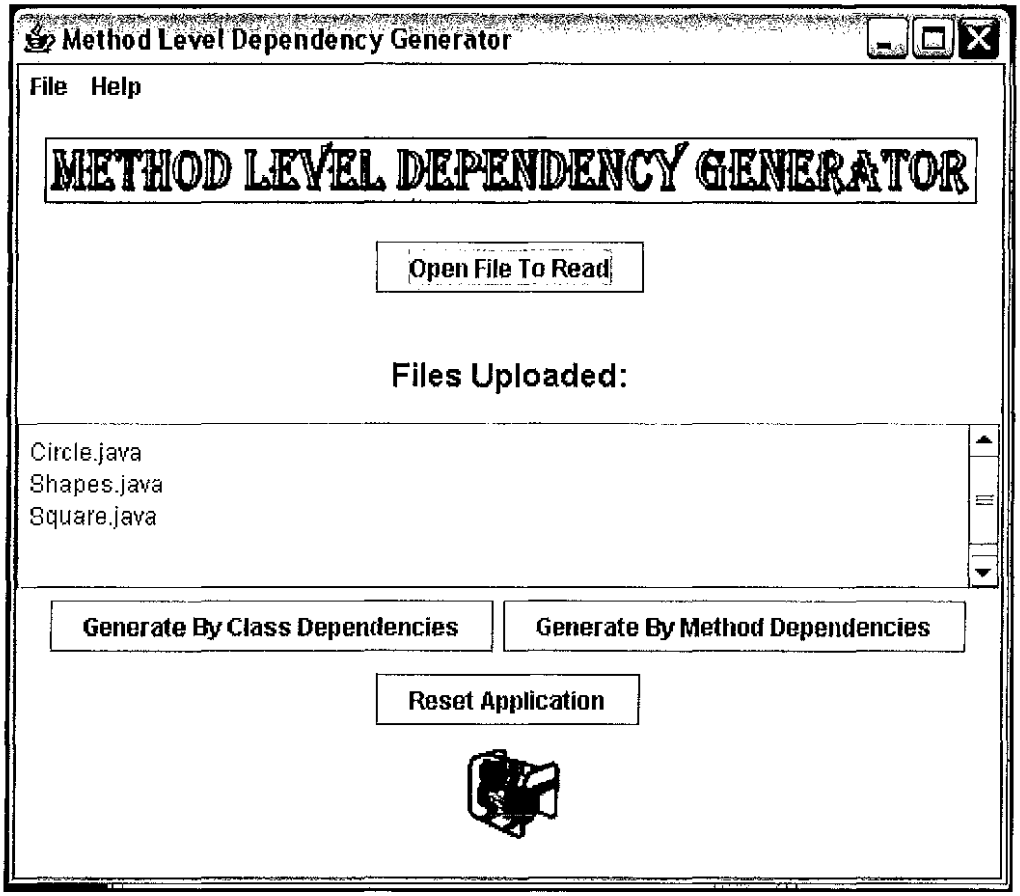
Look In: Test4

File Name: Circle.java

Files of Type: All Files

Figure 16: Method Level Dependency Generator File Selector

The file was uploaded and scanned for the various structural characteristics, indicating a class declaration, method declaration, class instantiation, or a method call. As a method call was found, the current class, current method, the class in which the called method was contained, and the called method name were all saved to the database. This was repeated for each file the user wished to read. The user was able to view a list of all files read, thus far, in the file list in the GUI. In order to quit the application, the user can select File on the menu bar and the Quit option. The Help option on the menu bar would be used to provide the user with help information. The Method Level Dependency Generator interface can be seen in Figure 17.



Circle.java
Shapes.java
Square.java

Figure 17: Method Level Dependency Generator

Once all files were evaluated, the user selected either the “Generate By Class Dependencies” button or the “Generate by Method Dependencies” button, to generate the desired diagrams to show the detailed dependencies for all the uploaded classes. The information can be cleared from the database by selecting the “Reset Application” button, in order to start clean again.

3.3 Framework Output

There are two diagrams that were generated by the Method Level Dependency Generator. These diagrams included a diagram by class showing the class and method calls from it, illustrated in Figure 18, and a diagram by method showing the methods that access it, illustrated in Figure 19.

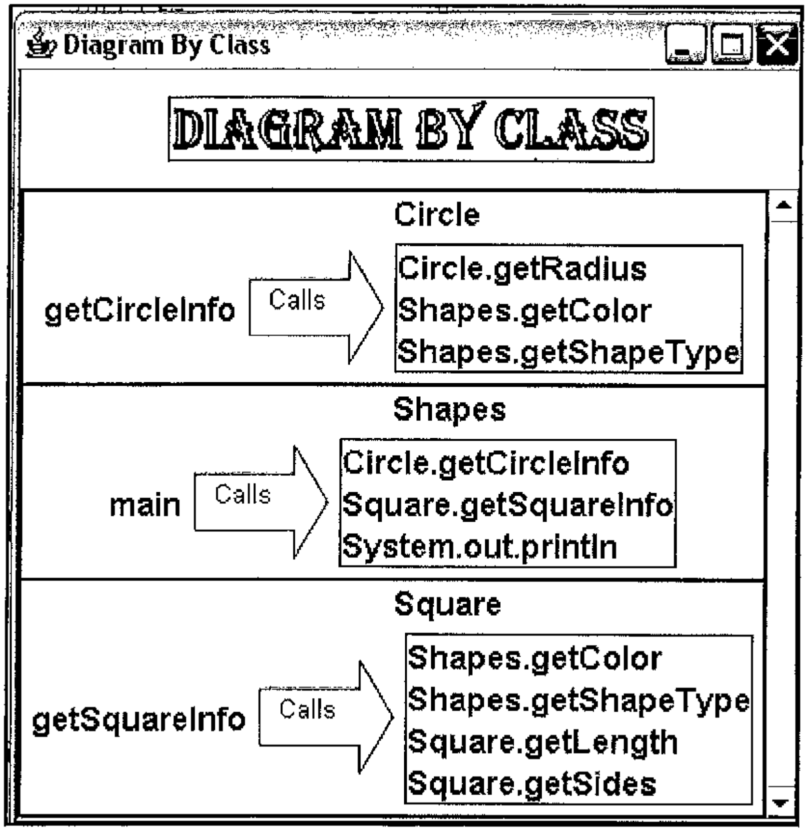
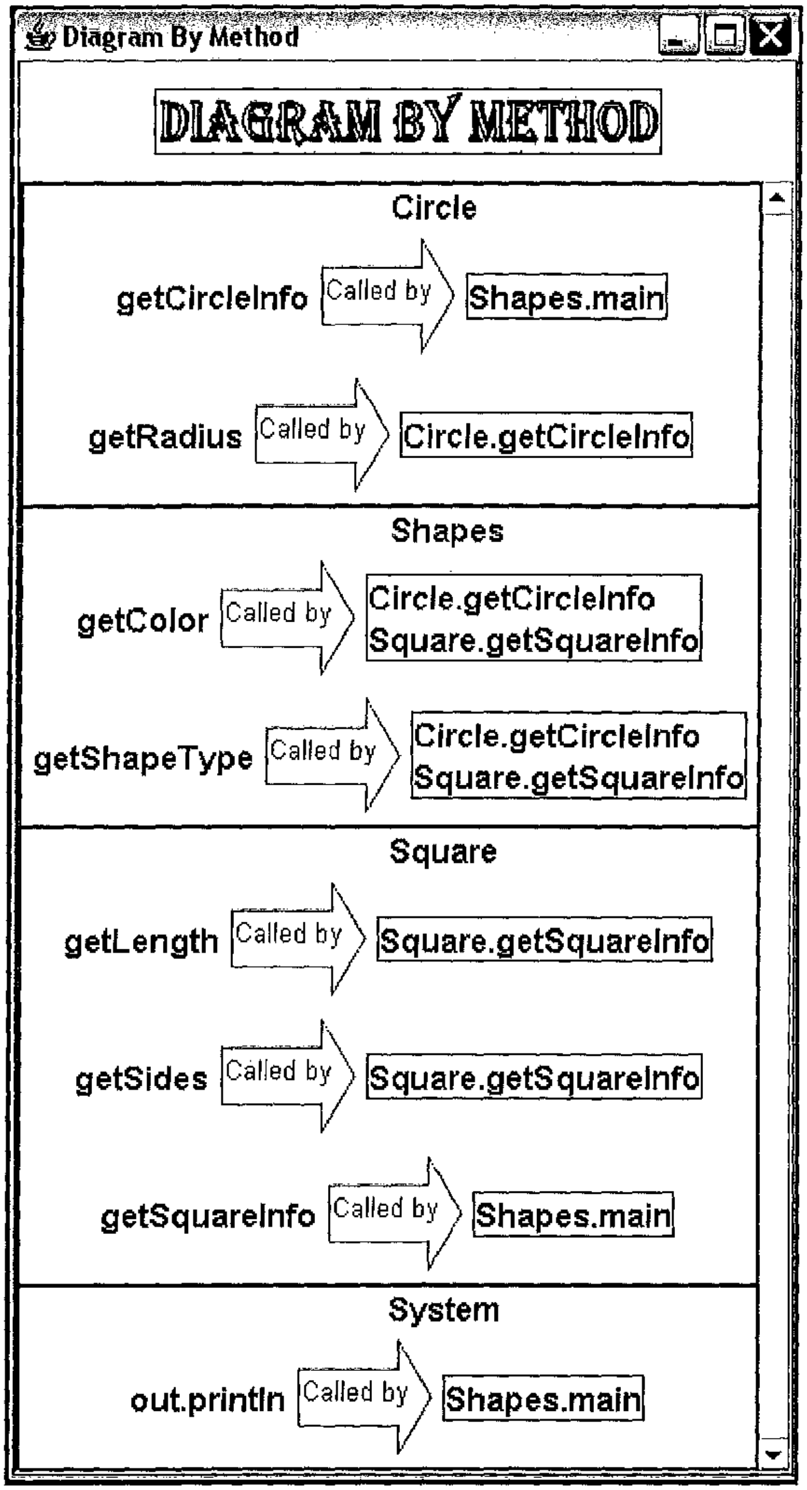


Figure 18: Generate By Class Dependencies



Circle

getCircleInfo Called by Shapes.main

getRadius Called by Circle.getCircleInfo

Shapes

getColor Called by Circle.getCircleInfo
Square.getSquareInfo

getShapeType Called by Circle.getCircleInfo
Square.getSquareInfo

Square

getLength Called by Square.getSquareInfo

getSides Called by Square.getSquareInfo

getSquareInfo Called by Shapes.main

System

out.println Called by Shapes.main

Figure 19: Generate By Method Dependencies

The “Diagram by Class” diagram (Figure 18) shows a representation of all classes, the methods that are in them, and what methods they depend on. For instance, this example contained three different classes, Circle, Shapes, and Square. The Circle class contained one method, getCircleInfo(), that has method dependencies. These method dependencies included Circle.getRadius(), Shapes.getColor(), and Shapes.getShapeType(). The Shape class contained one method, main(), that calls other methods, Circle.getCircleInfo(), Square.getSquareInfo(), and System.out.println(). Finally, the Square class had the method getSquareInfor() that called Shapes.getColor(), Shapes.getShapeType(), Square.getLength(), and Square.getSides(). From this diagram, the user was able to see by class what other classes and methods a change would potentially affect.

The “Diagram by Method” diagram (Figure 19) shows a representation of all methods and what methods depend on them. In the above example, the application being tested contained seven methods spread over three classes. The class Circle contained the methods getCircleInfo() called by Shapes.main() and getRadius() called by Circle.getCircleInfo(). The Shapes class contained two methods, getColor() and getShapeType(). Both of these methods were called by Circle.getCircleInfo() and Square.getSquareInfo(). The Square class is made up of getLength() called by Square.getSquareInfo(), getSides() called by Square.getSquareInfo(), and getSquareInfo() called by Shapes.main().

Upon completion of this new reverse engineering scheme, both method level dependency diagrams were compared with the existing diagrams generated by the other commonly-used approaches. This was accomplished by analyzing the results for each diagram. By examining output from the existing methods, along with output provided by this new reverse engineering framework, it was apparent that the new framework provided a greater level of detail. The provision of method level dependencies in combination with the output of existing tools should provide a more practical tool for software maintenance.

Chapter 4

RESULTS

The new reverse engineering framework provided for the display of method level dependencies, in addition to the diagrams of existing tools. The Method Level Dependency Generator was designed to read in Java source code input files as a source for the generation of the desired detailed diagrams. The test bed for this thesis contained many different test cases obtained from various sources, including some previous school projects. Each test case was made up of multiple class files, all varying in different characteristics, such as size and functionality. All test files contained, at the minimum, the essential information to retrieve the desired results, such as method calls and the respective method signatures. The important factor for this thesis was to present the method dependencies; thus, the test files focused on method calls. In order to test the new methodology, each test case was compiled and loaded into the reengineering framework. This generated the diagrams to display the lower level dependencies in the questionable application. The new method level dependency approach is beneficial when used in conjunction with existing software that shows high level dependencies. Many test cases were run and the results attest to this finding.

4.1 Test Case 1

The first test case consisted of four different classes. This application created multiple book objects and added the books to a library. Figure 20 shows the UML diagram, generated by Rational Rose®, which displays the different classes and methods within them, with their relationships to the other classes within the application. However, with just this diagram, it is difficult to see the method level dependencies; essentially, which methods really affect other methods. By using the new methodology, one was able to view the method level dependencies. Allowing the user to see, in particular, which potential maintenance efforts on one method may produce effects another method.

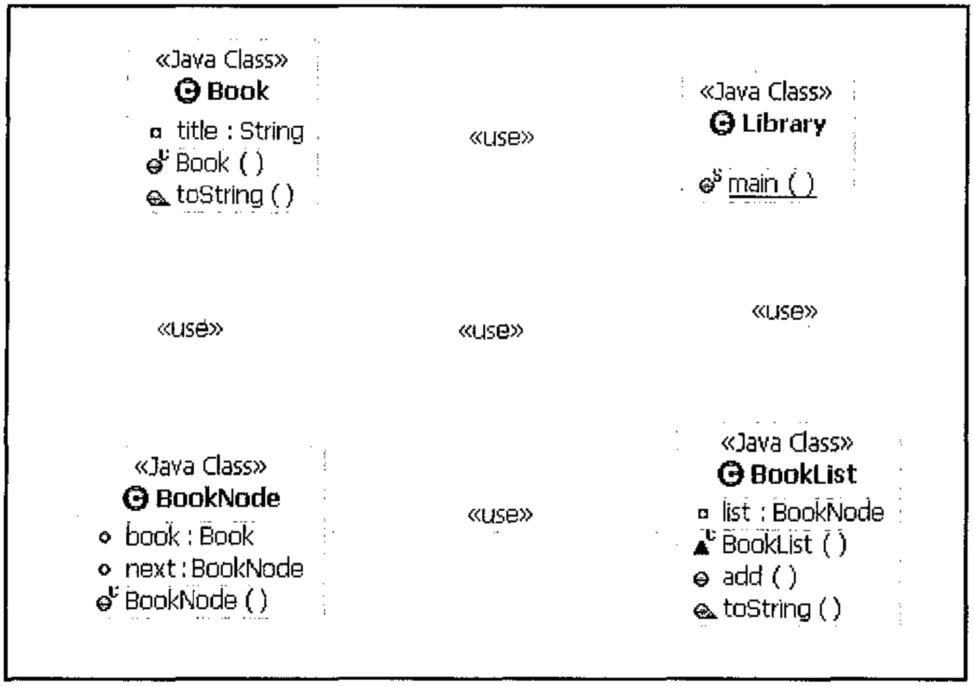


Figure 20: Test 1- Class Diagram

Figure 21, Diagram By Class showed the developer that the BookList class contained a toString() method that called the Book.toString() method and the Library class contained the main() method, which called both BookList.add() and System.out.println().

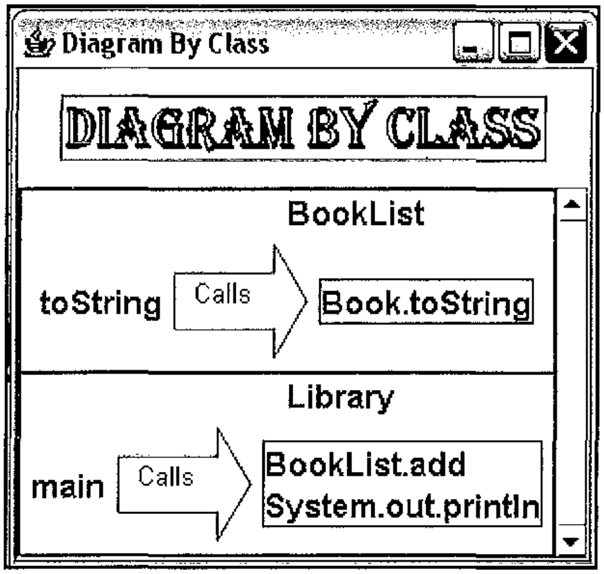


Figure 21: Test 1- Diagram By Class

Figure 22 illustrates the Diagram By Method functionality. Here, the user was informed the Book.toString() method was called by the BookList.toString() method, the BookList.add() method was called by Library.main(), and the System.out.println() method was also called by Library.main().

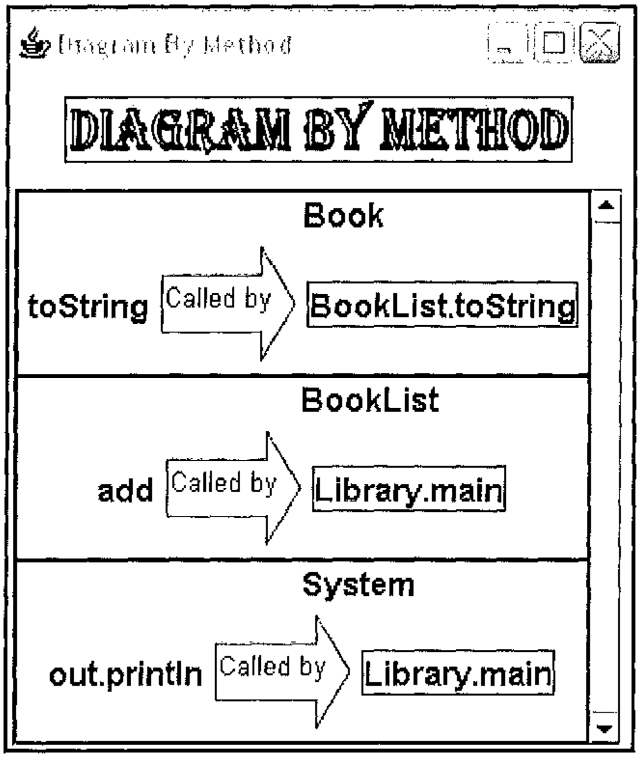


Figure 22: Test 1- Diagram By Method

4.2 Test Case 2

The second test case was taken from an accounting application. This system consisted of two classes that would create a bank account and then deposit funds, withdraw funds, and add interest to the accounts. Rational Rose® generated the UML class diagram depicting the two classes and their dependency on each other, as shown in Figure 23. Notice that the UML diagram does include the traditional UML dependency arrow. While this is helpful, it does not allow the user to really see any detailed level dependencies. However, when the UML dependency model was used with the reverse engineering framework, the user had a greater amount of information available to them, information ranging from architectural dependencies to detailed method-level dependencies, which could provide for a better understanding of the code.

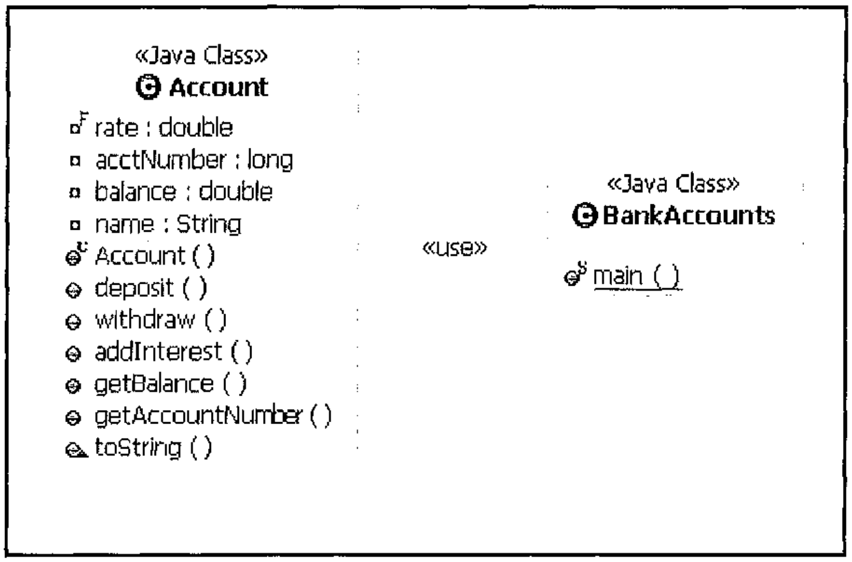


Figure 23: Test 2- Class Diagram

The Diagram By Class, in Figure 24 shows the Account class contained the methods deposit() and withdraw(), both of which invoked System.out.println(). The BankAccounts class contained a main() method, which called Account.addInterest(), Account.deposit(), Account.withdraw(), and System.out.println().

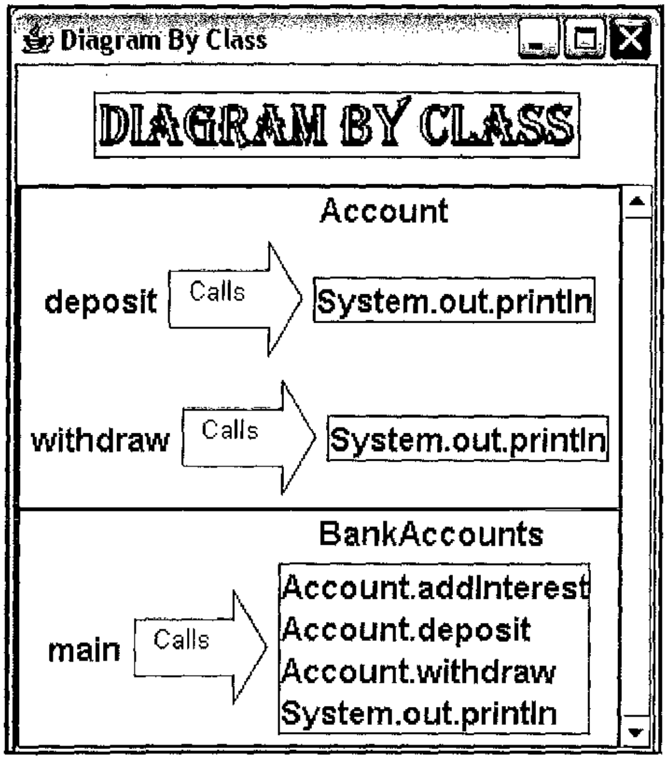


Figure 24: Test 2- Diagram By Class

Figure 25 showed the Diagram By Method, which focused on the methods and what methods depended upon them. For this test case, Account.addInterest(), Account.deposit(), and Account.withdraw() were all called by BankAccounts.main(). Any modifications to the System.out.println() could potentially have affected Account.deposit(), Account.withdraw(), and BankAccounts.main().

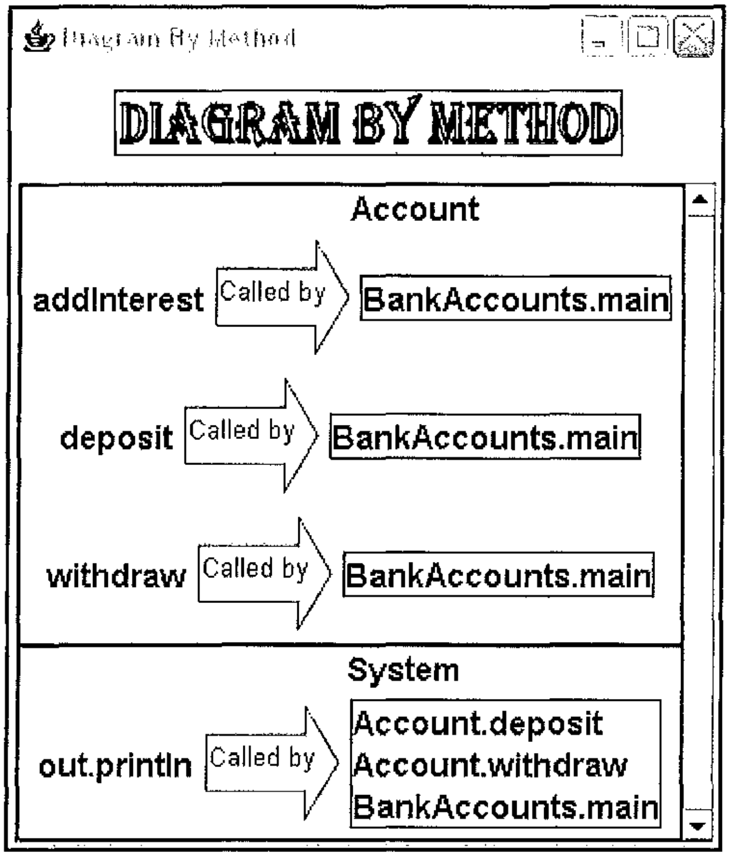


Figure 25: Test 2- Diagram By Method

4.3 Test Case 3

The third test case example was taken from a sports application. This system contained four different classes each having various methods, which provided helpful information about the application. The UML diagram, created by Rational Rose®, illustrated that the Basketball, Football, and Soccer classes were all related to the Sports class, shown in Figure 26. This was useful; however, when maintaining code, the developer will need more information about these dependencies. The new approach offers much more detailed information.

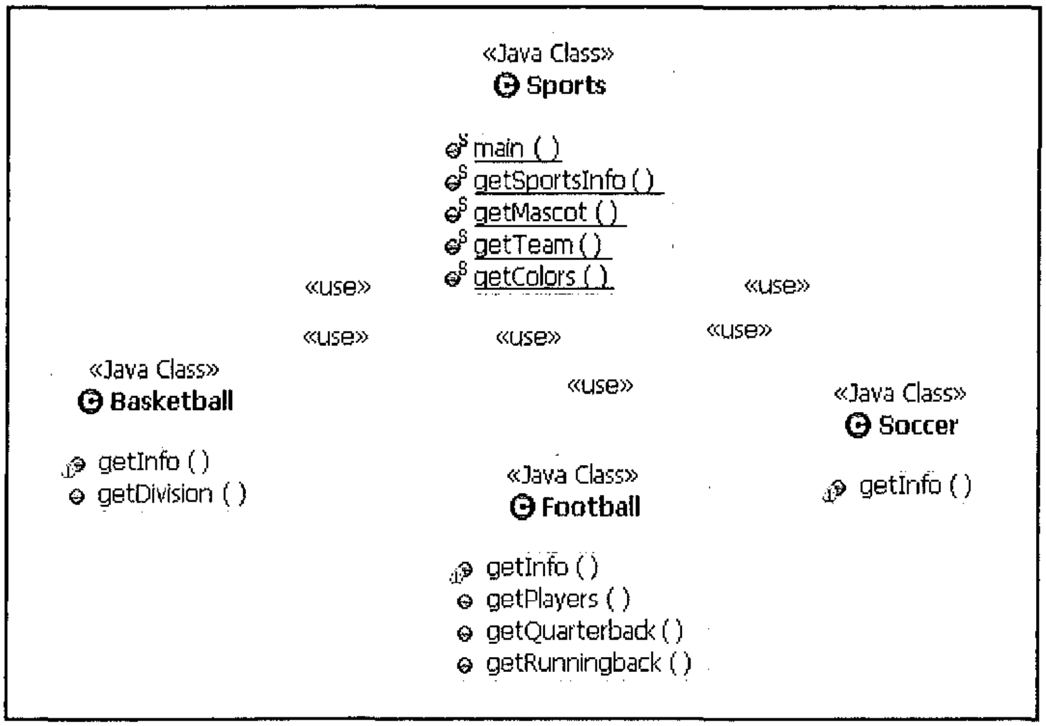


Figure 26: Test 3- Class Diagram

Figure 27 was the Diagram By Class. This diagram showed the different classes and what methods they call, or what methods they are dependent upon. The Basketball class' getInfo() method called Basketball.getDivision(), Sports.getColors(), Sports.getMascott(), and Sports.getTeam(). The Football class contained the method getInfo(), which depended on Football.getPlayers(), Sports.getColors(), Sports.getMascott(), and Sports.getTeam(); and, the method getPlayers(), which called Football.getQuarterback() and Football.getRunningback(). The third class, Soccer, had one method with dependencies, getInfo(), which called Sports.getColors(), Sports.getMascott(), and Sports.getTeam(). Finally, the Sports class had two methods which utilize other methods. The getSportsInfo(), which used Basketball.getInfo(), Football,getInfo(), and Soccer.getInfo(); and, the main() method, which used Sports.getSportsInfo() and System.out.println().

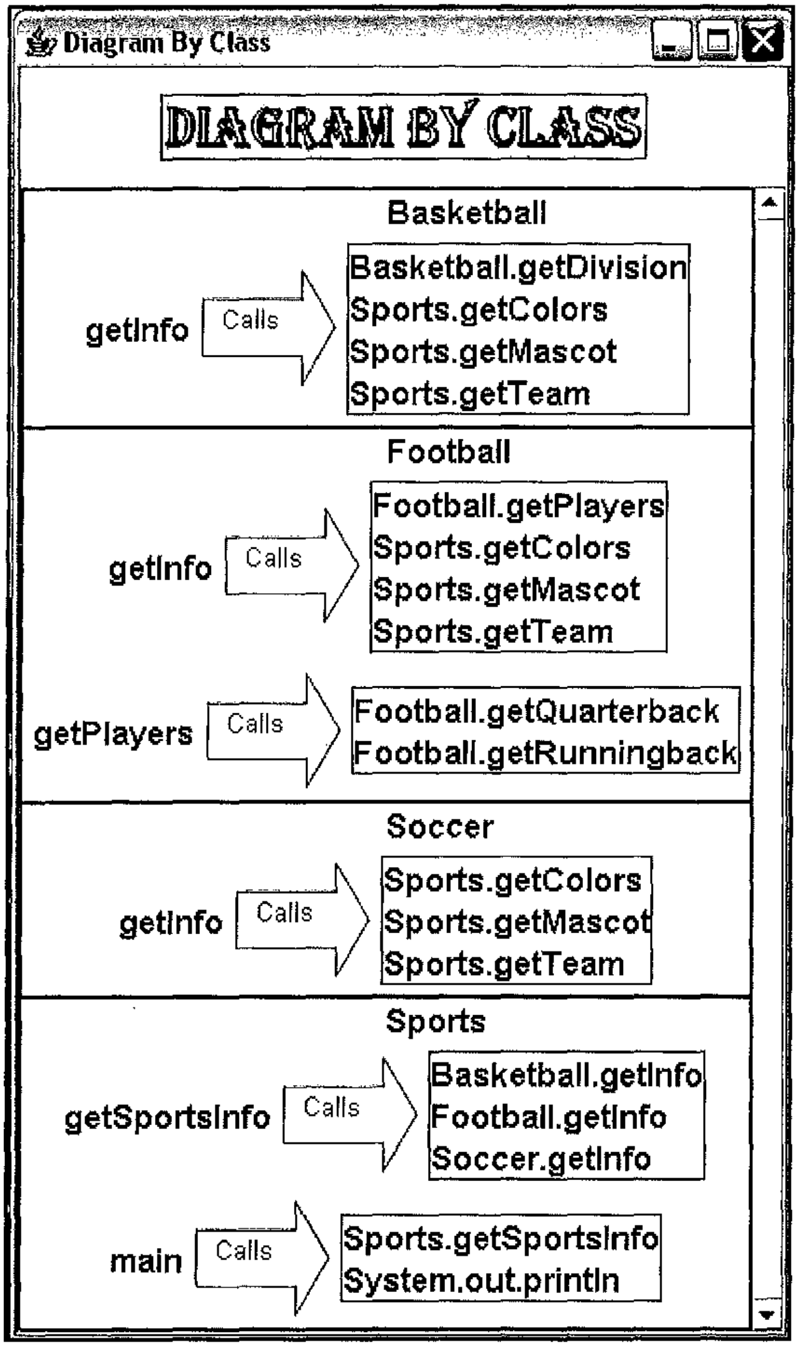


Figure 27: Test 3- Diagram By Class

Finally, Figure 28 shows the Diagram By Method, which presents all methods called by another method. By using this diagram, the user would know that any changes to Basketball.getDivision() could affect Basketball.getInfo() and any modifications to

Basketball.getInfo() could affect Sports.getSportsInfo(). The Football class contained three different method dependencies. The getPlayers() method was called by Football.getInfo(), the getQuarterback() method and getRunningback() method were both called by Football.getPlayers(). The Soccer.getInfo() method was called by only one other method, Sports.getSportsInfo(). The Sports class contained a few dependencies, including getColors(), getMascot(), and getTeam() methods, which were all called by Basketball.getInfo(), Football.getInfo(), and Soccer.getInfo(). The getSportsInfo() method was called by Sports.main(). Any changes to System.out.println() would only affect Sports.main().

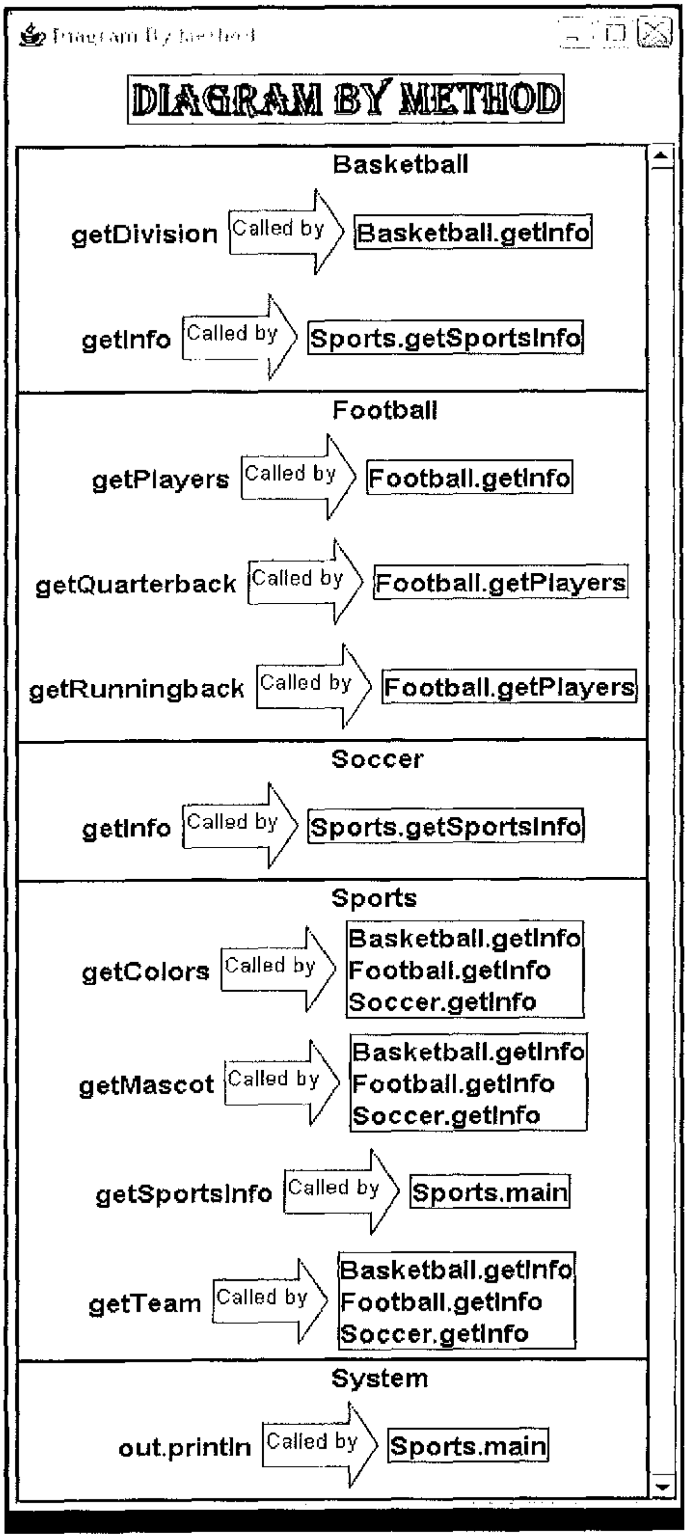


Figure 28: Test 3- Diagram By Method

4.4 Test Case 4

The final test case was a Tree application, which would allow the user to build a tree and then find nodes, insert nodes, delete nodes, get a node successor, traverse the node by pre-order, post-order, and in-order, and then display the tree. This program was made up of four different classes, each with a variety of methods to perform the desired functionality. All of these are illustrated in Figure 29, along with the class level dependencies, in the UML class diagram produced by Rational Rose®. When performing software maintenance, as usual, this would be beneficial, but not to the level a developer really needed. However, when the UML class diagram was used in conjunction with the method level dependency methodology, more information was available, which should make software maintenance more efficient.

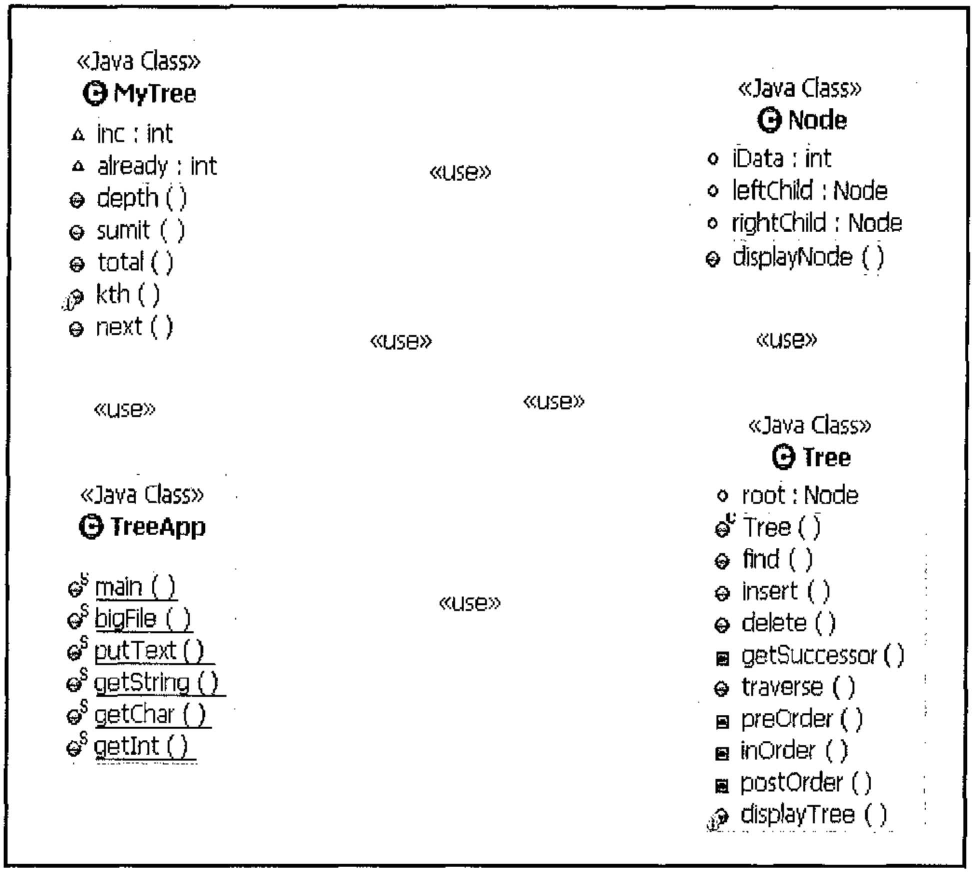


Figure 29: Test 4- Class Diagram

Figures 30 shows the Diagram By Class, illustrating all the classes and their method dependencies. The MyTree class contained the kth() method that called MyTree.next() and System.out.println() and the next() method that called MyTree.next() and Node.displayNode(). The displayNode() method found in the Node class had one dependency, System.out.println(). The Tree class had multiple method level dependencies. The delete() method called Tree.getSuccessor(). The displayTree() method requested the Stack.isEmpty(), Stack.pop(), Stack.push(), System.out.print(), and System.out.println(). The inOrder() method made a call to both Node.displayNody() and itself, Tree.inOrder(). The same was true for the postOrder() method and preorder() method. The method postOrder() called

Node.displayNode() and itself, Tree.postOrder(). The method preorder() also called Node.displayNode() and itself, Tree.preOrder(). The final method dependency in the Tree class was traverse() which, called System.out.println(), Tree.inOrder(), Tree.postOrder(), and Tree.preOrder().

The TreeApp class contained quite a few method calls. The TreeApp.bigFile() method directed the applications functionality. This method called BufferedReader.close(), BufferedReader.readLine(), Integer.parseInt(), MyTree.depth(), MyTree.kth(), MyTree.sumit(), MyTree.total(), Node.displayNode(), StringTokenizer.hasMoreTokens(), StringTokenizer.nextToken(), System.out.println(), Tree.delete(), Tree.displayTree(), Tree.find(), Tree.insert(), Tree.traverse(), TreeApp.getChar(), TreeApp.getInt(), and TreeApp.putText(). The getChar() method in TreeApp called the String.charAt() method and TreeApp.getString(). Similar to that, the TreeApp.getInt() method called Integer.parseInt() and TreeApp.getString(). The method getString() found in TreeApp only made one call to BufferedReader.readLine(). The main() method called both System.out.println() and TreeApp.bigFile(). The last method was putText(), which called System.out.flush() and System.out.println().

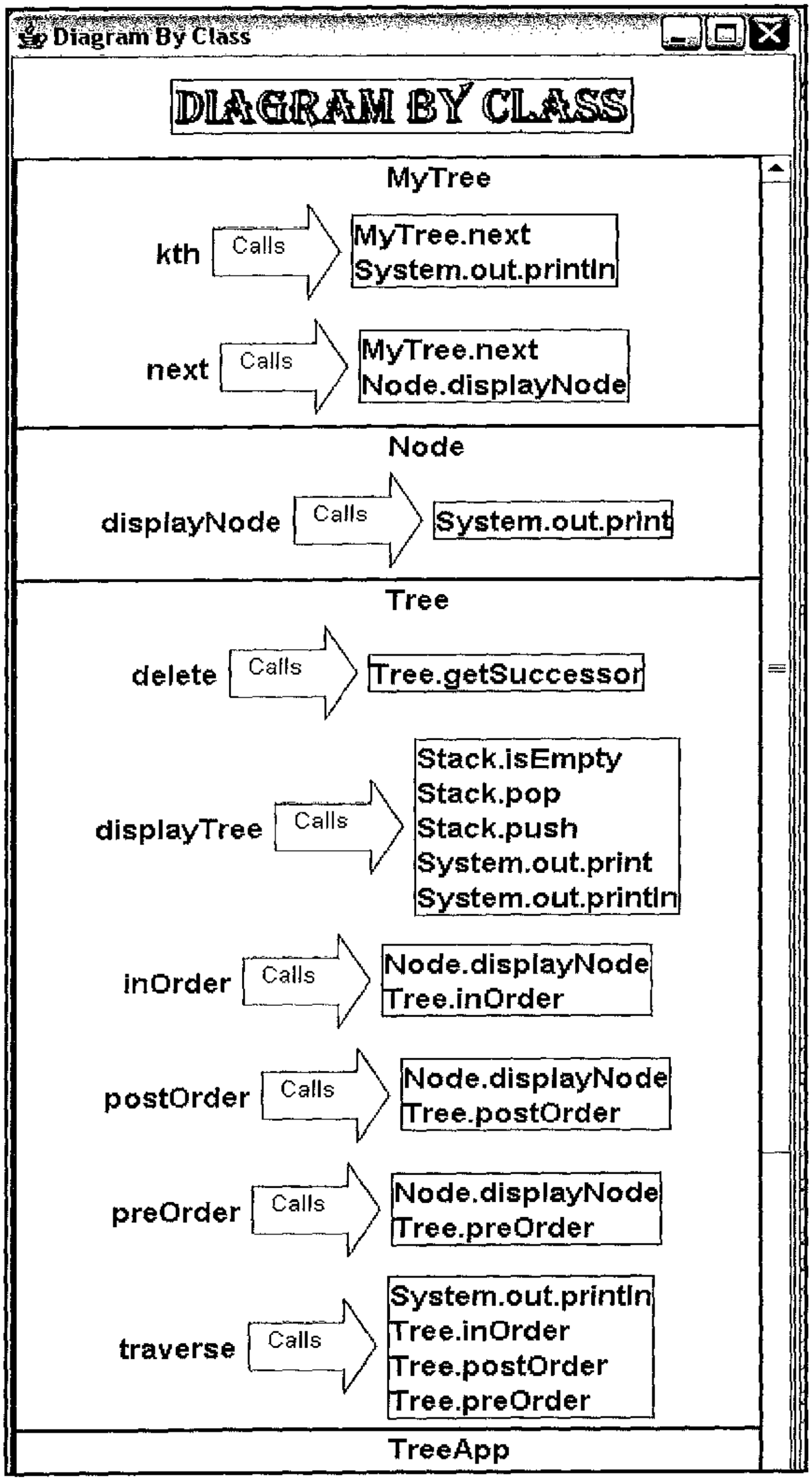


DIAGRAM BY CLASS

MyTree

kth Calls
MyTree.next
System.out.println

next Calls
MyTree.next
Node.displayNode

Node

displayNode Calls
System.out.print

Tree

delete Calls
Tree.getSuccessor

displayTree Calls
Stack.isEmpty
Stack.pop
Stack.push
System.out.print
System.out.println

inOrder Calls
Node.displayNode
Tree.inOrder

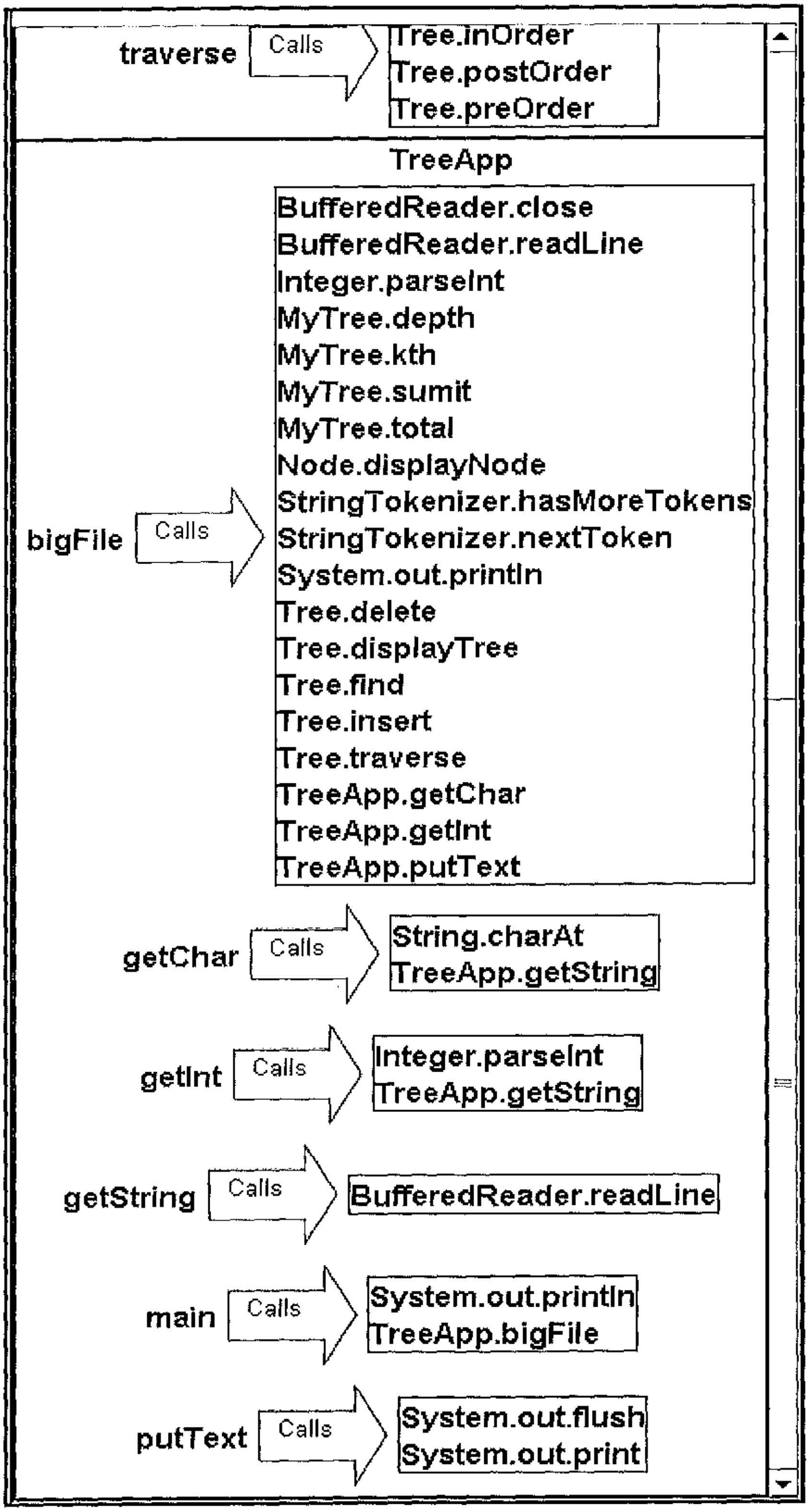
postOrder Calls
Node.displayNode
Tree.postOrder

preOrder Calls
Node.displayNode
Tree.preOrder

traverse Calls
System.out.println
Tree.inOrder
Tree.postOrder
Tree.preOrder

TreeApp

Figure 30: Test 4- Diagram By Class



traverse	Tree.inOrder Tree.postOrder Tree.preOrder

TreeApp

bigFile	BufferedReader.close BufferedReader.readLine Integer.parseInt MyTree.depth MyTree.kth MyTree.sumit MyTree.total Node.displayNode StringTokenizer.hasMoreTokens StringTokenizer.nextToken System.out.println Tree.delete Tree.displayTree Tree.find Tree.insert Tree.traverse TreeApp.getChar TreeApp.getInt TreeApp.putText
getChar	String.charAt TreeApp.getString
getInt	Integer.parseInt TreeApp.getString
getString	BufferedReader.readLine
main	System.out.println TreeApp.bigFile
putText	System.out.flush System.out.print

Figure 30 - *continued*

The Diagram By Method functionality is displayed in Figure 31. This diagram shows, by method, where those methods were utilized throughout the application.

The BufferedReader class contained the methods close(), which was called by TreeApp.bigFile() and the method readLine(), which was also called by TreeApp.bigFile() and TreeApp.getString(). The parseInt() method in the Integer class was called by both methods bigFile() and getInt() in the TreeApp class.

The MyTree class included the methods depth(), kth(), sumit(), and total(), which were all called by TreeApp.bigFile(). TreeApp.next() was called by MyTree.kth() and MyTree.next().

The Node class only had one method dependency, found in the displayNode() method, which was called by MyTree.next(), Tree.inOrder(), Tree.postOrder(), Tree.preOrder(), and TreeApp.bigFile(). The methods isEmpty(), pop(), and push(), all from the Stack class, were called by Tree.displayTree().

The String class contained the charAt() method, which was used by TreeApp.getChar().

The StringTokenizer class contained both the hasMoreTokens() and nextToken(), both called by TreeApp.bigFile().

The System class contained multiple method dependencies. The out.flush() method was utilized by TreeApp.putText(), the out.print() method from Node.displayNode(), Tree.display.Tree(), and TreeApp.putText(), and the method out.println() from MyTree.kth(), Tree.displayTree(), Tree.traverse(), TreeApp.bigFile(), and TreeApp.main().

The Tree class contained a few methods that were all called by TreeApp.bigFile(), including Tree.delete(), Tree.displayTree(), Tree.find(), Tree.insert(), and Tree.traverse(). The method getSuccessor() was called by Tree.delete(). Finally, the methods inOrder(), postOrder(), and preorder() were all called by themselves and Tree.traverse().

The last class in this example was the TreeApp class. This class contained the method bigFile(), which was called by TreeApp.main(). The methods getChar(), getInt(), and putText() were all utilized by TreeApp.bigFile() and the method getString() by TreeApp.getChar() and TreeApp.getInt().

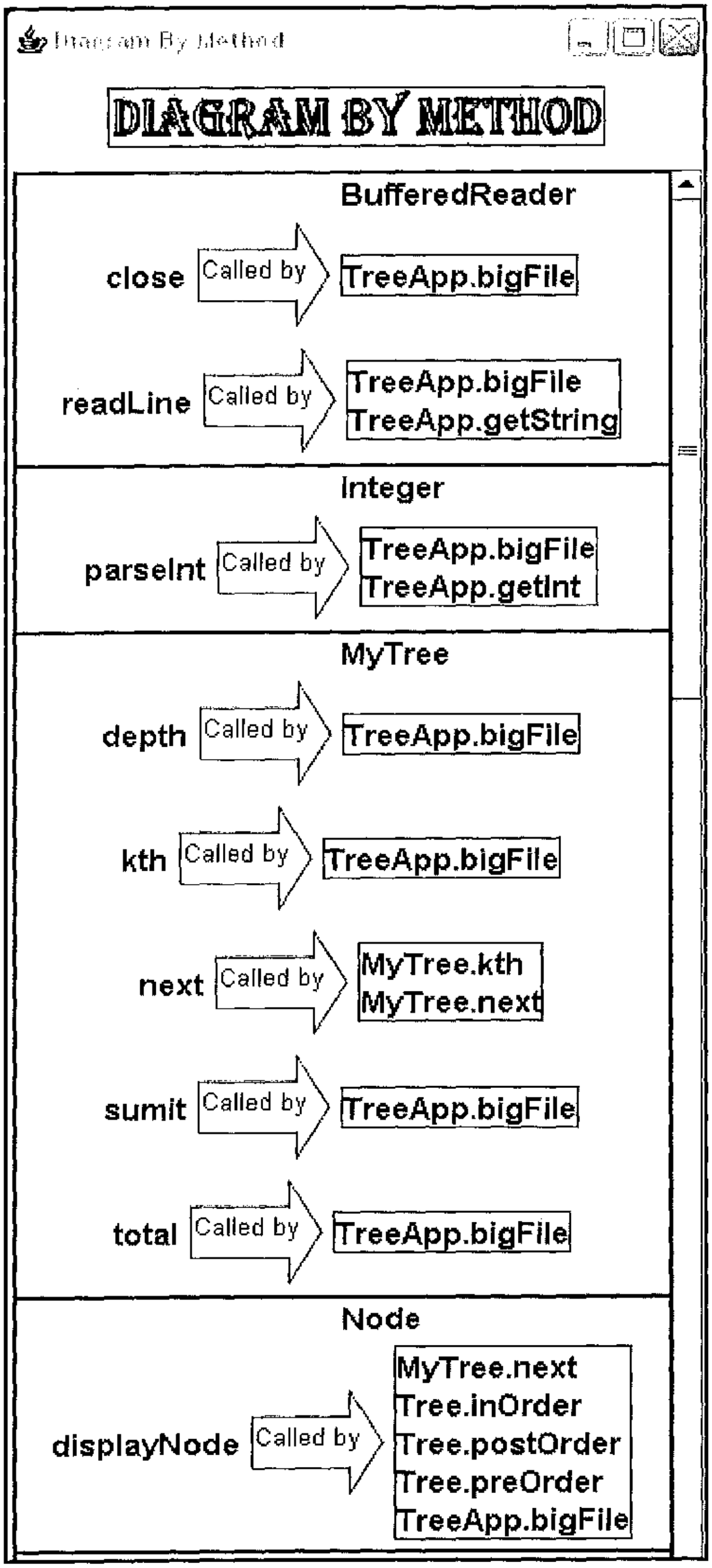


Diagram By Method

DIAGRAM BY METHOD

BufferedReader

close Called by TreeApp.bigFile

readLine Called by TreeApp.bigFile
TreeApp.getString

Integer

parseInt Called by TreeApp.bigFile
TreeApp.getInt

MyTree

depth Called by TreeApp.bigFile

kth Called by TreeApp.bigFile

next Called by MyTree.kth
MyTree.next

sumit Called by TreeApp.bigFile

total Called by TreeApp.bigFile

Node

displayNode Called by MyTree.next
Tree.inOrder
Tree.postOrder
Tree.preOrder
TreeApp.bigFile

Figure 31: Test 4- Diagram By Method

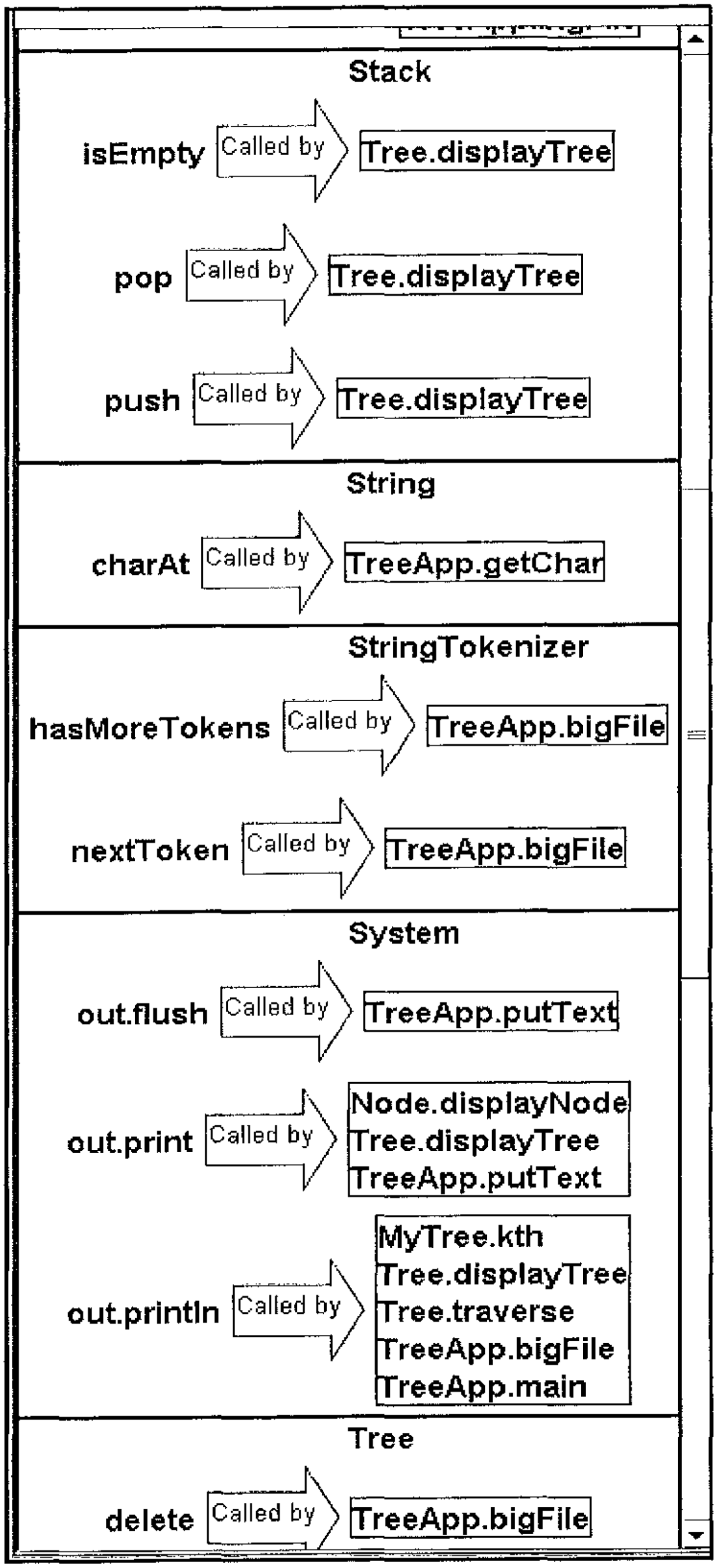


Figure 31 - *continued*

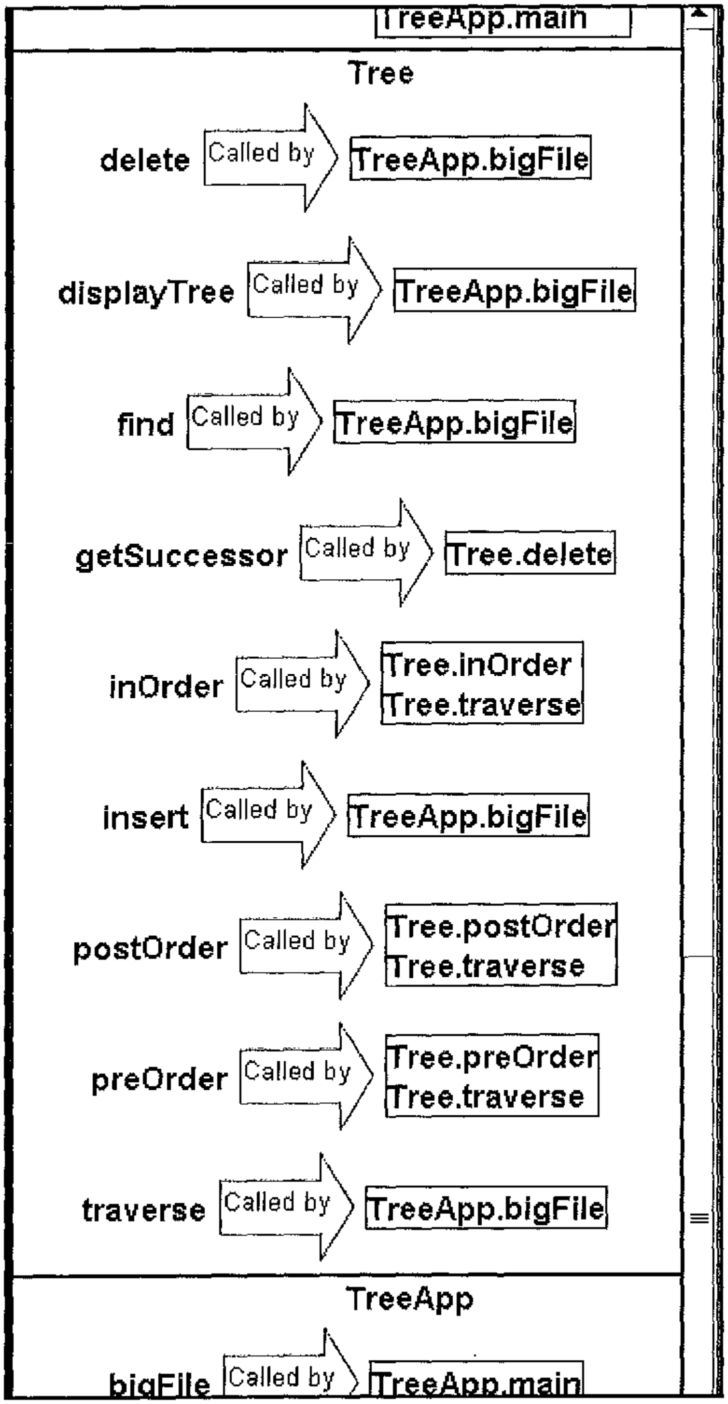
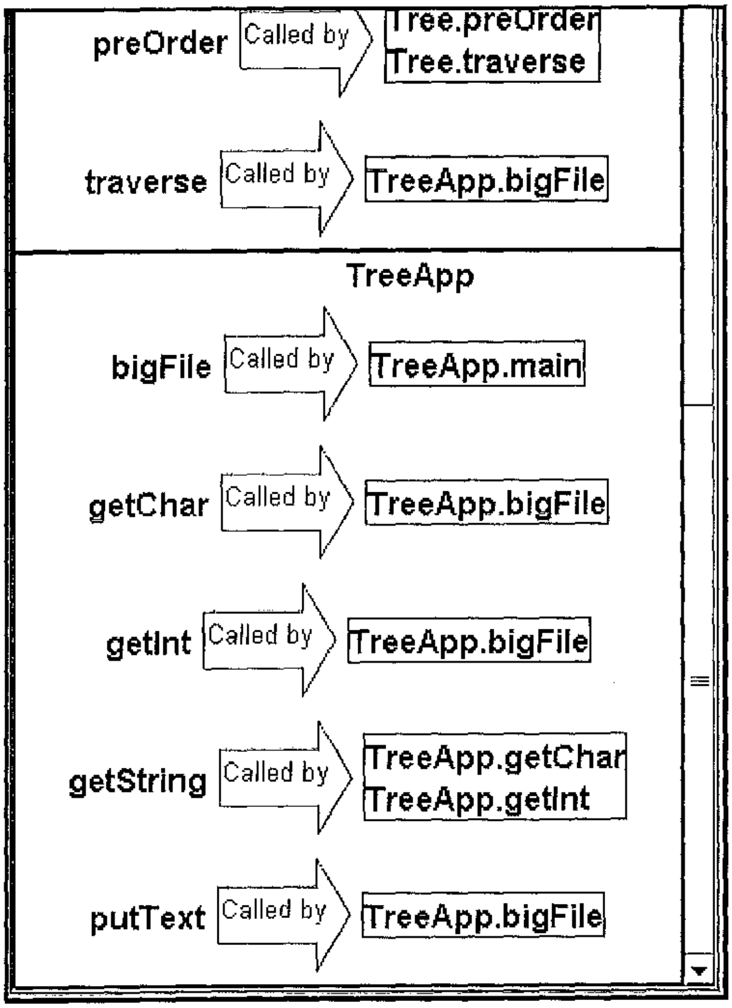


Figure 31 - *continued*



preOrder Called by Tree.preOrder
Tree.traverse

traverse Called by TreeApp.bigFile

TreeApp

bigFile Called by TreeApp.main

getChar Called by TreeApp.bigFile

getInt Called by TreeApp.bigFile

getString Called by TreeApp.getChar
TreeApp.getInt

putText Called by TreeApp.bigFile

Figure 31 - *continued*

Chapter 5

CONCLUSION

5.1 Analysis

The test cases provided for some of the more interesting examples of all the scenarios analyzed. The test suite showed the variety of test files that were studied. The test cases varied in complexity ranging from a simple test, such as test case one, to more difficult test cases, as in test case four. Results clearly indicate the comprehensive nature of the framework that includes, not only useful UML class diagrams, but the essential addition of method level dependencies.

By viewing these results it is clear the new framework, consisting of detailed method level dependencies in conjunction with higher level class diagrams, is a useful methodology for undertaking real world software maintenance. As each test case was evaluated, the framework was found to reliably produce lower level dependencies among complex Java methods. Diagrams produced within the framework provide a quick visual artifact of method level detail within specific applications. The reliability of maintenance activities should be much improved by the use of this framework in the workplace.

Each of the test cases demonstrated the use of the new framework that provides the software practitioner with a view of the source code characterized by a lower level of granularity. By examining the results for each scenario, the developer may readily observe the results of using this new framework. The results demonstrate how the UML class diagram provided for high level, architectural information of the application. However, this information alone leaves much to be desired regarding application specific logic and detail, which is where most code changes (and errors) occur. Both of the method level dependency diagrams assist the developer with a more detailed view of dependencies in code. In particular, the Diagram By Class facility indicates all methods each of the instance methods call, thus providing a map to other services provided by other methods in other classes. To complement the Diagram By Class facility, the Diagram By Method presented for each method in a class those methods in other classes that have dependencies upon the particular method. In summary, UML class diagrams, supplemented with diagrams by class and by method, provide a comprehensive framework to assist the software maintenance practitioner.

As a software engineer, it often feels as if there is simply not enough time in a day to get the job done. The reverse engineering framework has the potential to expedite many of the activities of those engaged in software development. This new method dependency approach provides a very practical, lower level of granularity that should be useful to professionals in the workplace. By coupling this new approach with existing software that generates UML diagrams with their higher level architectural descriptions of collaborating classes, the practitioner now posses a comprehensive methodology that addresses both the architectural class dependencies and other class relationships, along with a more detailed analysis of application design and code central to modern day development and maintenance needs.

Method level detail provides a higher degree of assurance in reconciling a myriad of maintenance duties in the workplace. While UML class diagrams are very helpful in displaying class relationships, the additional detailed information provided by this method level generator completes a comprehensive strategy, which should provide for significantly improved software maintenance efforts.

5.2 Future Work

There are several opportunities for future work, which may extend the utility of this framework and provide additional workplace value to software engineering practitioners.

The Method Level Dependency Generator was developed using Java. This framework recognized various ways to declare new classes, new methods, and method calls throughout various files loaded into the system. However, the Java programming language is quite complex, therefore, some potential enhancements exist.

The framework could be modified to be more robust and handle the entire range of Java syntax, such as recognizing every way an object can be instantiated. Some known issues, not yet accounted for in the Method Level Dependency Generator, include the capability to recognize creating an object and instantiating it separately and to recognize multiple functions within a line, such as declaring an object within a method call.

Similarly, the framework is not set up to account for implementing interfaces in Java. This is because a class can implement multiple interfaces. With the current design of the generator, when the application encounters a method call, it will not know for certain if the method is found in the current file or in one of the interfaces.

The new approach could also be improved by integrating this generator in with the existing technology for developing a class diagrams, such as Rational Rose®, Eclipse®, or jGRASP®. This could be accomplished various ways, such as, when the user views the class diagram, a provision could be made to click on a class to view the method dependencies.

REFERENCES

Print Publications:

[Ali05]
Ali, Muhammad Raza, "Why Teach Reverse Engineering," ACM SIGSOFT Software Engineering Notes, Volume 30, Issue 4; pp. 1-4, July 2005.

[Buss91]
Buss, Erich and John Henshaw, "A Software Reverse Engineering Experience," Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative Research CASCON '91, pp. 55-72, October 1991.

[Chen05]
Chen, Zhixiong and Delia Mars, "Experiences with Eclipse IDE in Programming," Consortium for Computing Sciences in Colleges, pp. 104-112, 2005.

[Demeyer00]
Demeyer, Serge, Stephane Ducasse, and Oscar Nierstrasz, "Finding Refactorings via Change Metrics," ACM SIGPLAN Notices, Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA '00, Volume 35, Issue 10, pp. 166-177, October 2000.

[Ebner02]
Ebner, Gerald and Hermann Kaindl, "Tracing All Around in Reengineering," IEEE Software, pp. 70-77, May/June 2002.

[El-Ramly06]
El-Ramly, Mohammad, "Experience in Teaching a Software Reengineering Course," Proceeding of the 28th international conference of Software Engineering ICSE '06, pp. 699-702, May 2006.

[Halsted02]
Halsted, Kari L. and James H. Roberts., "Eclipse Help System: An Open Source User Assistance Offering," Proceedings of the 20th annual international conference of Computer Documentation SIGDOC '02, pp. 49-59, October 2002.

[Merdes06]
Merdes, Matthias and Dirk Dorsch, "Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development," Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java PPPJ '06, pp. 125-134, August 2006.

[Muller93]
Muller, Hausi A, Scott R. Tilley, and Kenny Wong, "Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project." Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering Volume 1 CASCON '93, pp. 217-226, October 1993.

[Muller00]
Muller, Hausi, Jens Jahnke, Dennis Smith, Margaret-Anne Storey, Scott Tilley, and Kenny Wong, "Reverse Engineering: A Roadmap," Proceedings of the Conference on the Future of Software Engineering ICSE '00, pp. 47-60, May 2000.

[Muller97]
Muller, Hausi, "Reverse Engineering Strategies for Software Migration." Proceedings of the 19th international conference on Software Engineering ICSE '97, pp. 659-660, May 1997.

[Newcomb95]
Newcomb, Philip, "Web-Based Business Process Reengineering," IEEE Software, pp. 116-118, November 1995.

[Niestrasz04]
Nierstrasz, Oscar and Serge Demeyer, "Object-Oriented Reengineering Patterns," Proceedings of the 26th International Conference on Software Engineering (ICSE '04), pp. 1-8, 2004.

[Nierstrasz05]
Nierstrasz, Oscar, Stephane Ducasse, and Tudor Girba, "The Story of Moose: an Agile Reengineering Environment," ACM SIGSOFT Software Engineering Notes, Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering ESEC/FSE-13, Volume 30, Issue 5, pp. 1-10, September 2005.

[Rountev05]
Rountev, Atanas and Beth Harkness Connell, "Object Naming Analysis for Reverse-Engineered Sequence Diagrams," Proceedings of the 27th international conference on Software Engineering ICSE '05, pp. 254-263, May 2005.

[Sneed95]
Sneed, Harry, "Planning the Reengineering of Legacy System," IEEE Software, pp. 24-34, January 1995.

[Tilley01]
Tilley, Scott, and Shihong Huang, "Evaluating the Reverse Engineering Capabilities of Web Tools for Understanding Site Content and Structure: A Case Study," Proceedings of the 23rd International Conference on Software Engineering ICSE '01, pp. 514-523, July 2001.

[Tomic94]
Tomic, Marijana, "A Possible Approach to Object-Oriented Reengineering of Cobol Programs," ACM SIGSOFT Software Engineering Notes, Volume 19, Issue 2, pp. 1-6, April 1994.

[Tonella05]
Tonella, Paolo, "Reverse Engineering of Object Oriented Code," Proceedings of the 27th International Conference on Software Engineering ICSE '05, pp. 724-725, May 2005.

Electronic Sources:

[Auburn University07]
Auburn University, "jGRASP: An Integrated Development Environment with Visualizations for Improving Software Comprehensibility," Auburn University, http://www.eng.auburn.edu/department/cse/research/grasp/, last accessed 2007.

[Bongfiglio02]
Bonfiglio, Fransesco, "Reverse Engineering Legacy Code with Rational Rose," Rational Software, http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/apr02/ReverseEngineeringApr02.pdf, last accessed 2002.

[Cross06]
Cross, James II and Dean Hendrix, "Workshop jGRASP: An Integrated Development Environment with Visualizations for Teaching Java in CS1, CS2, and Beyond," IEEE, http://fie.engrng.pitt.edu/fie2006/papers/1837.pdf, last accessed 2006.

[Eclipse07]
Eclipse, "Eclipse - an open development platform," Eclipse, http://www.eclipse.org/, last accessed 2007.

[IBM07]
IBM, "Rational Rose," IBM, http://www-306.ibm.com/software/awdtools/developer/rose/index.html, last accessed 2007.

[NetBeans07]
NetBeans, "NetBeans IDE 5.5.1," NetBeans, http://www.netbeans.org/, last accessed 2007.

[Thao06]
Thao, Tom, Chaymous Klang, and Ben Talberg, "JGRASP: a code analyzing tool," http://www-users.cs.umn.edu/~dliang/5802reports/06/Thao/jgrasp.pdf, last accessed 2006.

APPENDIX A

Source Code: Constants.java

```
public class Constants
{
      public static final String currentClass = "(class)(\\s)";
      public static final String currentMethod =
            "(\\w+)(\\[\\])*(\\s)(\\w+)(\\s)*(\\()";
      public static final String Objects =
                  "(new)(\\s)(\\w+)(\\s)*(\\()";
      public static final String StringObjects =
                  "(String)(\\s)(\\w+)(\\s)*(=)";
      public static final String Methods = "(\\.)*(\\w+)(\\s)*(\\()";
      public static final String arrow2 = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/arrow2.jpg";
      public static final String arrow3 = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/arrow3.jpg";
      public static final String title = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/Title.jpg";
      public static final String generator = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/generator.jpg";
      public static final String classTitle = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/ClassTitle.jpg";
      public static final String methodTitle = "C:/Documents and
            Settings/lehays/workspace/Thesis/images/MethodTitle.jpg";
}
```

APPENDIX B

Source Code: DatabaseMethods.java

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

public class DatabaseMethods
{
      public Connection connection = null;
      public void getConnection() throws SQLException
      {
            try
            {
                  // Load the JDBC driver
                  // MySQL MM JDBC driver
                  String driverName = "org.gjt.mm.mysql.Driver";
                        Class.forName(driverName);
                  // Create a connection to the database
                  String serverName = "localhost";
                  String mydatabase = "thesis";
                        String url = "jdbc:mysql://" + serverName +
                        "/" +
                        mydatabase; // a JDBC url
                  String username = "root";
                  String password = "rigsby";
                  connection = DriverManager.getConnection(url,
                        username,
                        password);
            }
            catch (ClassNotFoundException e)
            {
                  System.out.println("could not find the database
                        driver");
            }
            catch (SQLException e)
            {
                  System.out.println("could not connect to the
                        database");
            }
            finally
            {
            }
      }

      public void insertCode(String CurrentClass, String
CurrentMethod,
            String CalledClass, String CalledMethod) throws
            SQLException
      {
```

```
        PreparedStatement Stat = null;
        try
        {

                if (connection == null)
                {
                        getConnection();
                }

                String sSQLString =
                        " INSERT INTO code (CurrentClass,
                        CurrentMethod, CalledClass, CalledMethod)
                        values(?,?,?,?)";

                Stat = connection.prepareStatement(sSQLString);

                Stat.setString(1, CurrentClass);
                Stat.setString(2, CurrentMethod);
                Stat.setString(3, CalledClass);
                Stat.setString(4, CalledMethod);

                if (Stat.executeUpdate() == 0)
                {
                        System.out.println("did not insert");
                }
        }
        catch (SQLException e)
        {
                System.out.println("SQL Exception" + e);
        }
        finally
        {
                Stat.close();
                connection.close();
        }
}

public void resetApplication() throws SQLException
{
        PreparedStatement Stat = null;
        try
        {
                if (connection == null)
                {
                        getConnection();
                }

                String sSQLString = "delete from code";

                Stat = connection.prepareStatement(sSQLString);

                if (Stat.executeUpdate() == 0)
                {
                        System.out.println("did not clear out the
                                database");
                }
```

```
        }
        catch (SQLException e)
        {
              System.out.println("SQL Exception: " + e);
        }
        finally
        {
              Stat.close();
              connection.close();
        }
  }

  public HashMap getDiagramInfoByClass() throws SQLException
  {

        PreparedStatement Stat = null;
        ResultSet ResultSet = null;
        HashMap Results = new HashMap();
        try
        {
              if (connection == null)
              {
                    getConnection();
              }

              String sSQLString =
                    "SELECT distinct * FROM code order by
                    CurrentClass, CurrentMethod;";

              Stat = connection.prepareStatement(sSQLString);
              ResultSet = Stat.executeQuery();

              int counter = 1;
              while (ResultSet.next())
              {
                    Results.put("CurrentClass" +
                          counter,ResultSet.getString("CurrentCla
                    ss
                          "));
                    Results.put("CurrentMethod" +
                          counter,ResultSet.getString("CurrentMet
                          hod"));
                    Results.put("CalledClass" +
                          counter,ResultSet.getString("CalledClas
                          s"));
                    Results.put("CalledMethod" +
                          counter,ResultSet.getString("CalledMeth
                    od
                          "));
                    counter++;
              }
        }
        catch (SQLException e)
        {
              System.out.println("SQL Exception" + e);
        }
```

```
        finally
        {
              Stat.close();
              ResultSet.close();
              connection.close();
        }
        return Results;
  }
  public HashMap getDiagramInfoByMethod() throws SQLException
  {
        PreparedStatement Stat = null;
        ResultSet ResultSet = null;
        HashMap Results = new HashMap();
        try
        {
              if (connection == null)
              {
                    getConnection();
              }
              String sSQLString =
                    "SELECT distinct * FROM code order by
                    CalledClass, CalledMethod ";
                    Stat =
              connection.prepareStatement(sSQLString);
              ResultSet = Stat.executeQuery();
              int counter = 1;
              while (ResultSet.next())
              {
                    Results.put("CurrentClass" +
                          counter,ResultSet.getString("CurrentCla
                          ss"));
                    Results.put("CurrentMethod" +
                          counter,ResultSet.getString("CurrentMet
                          hod"));
                    Results.put("CalledClass" +
                          counter,ResultSet.getString("CalledClas
                          s"));
                    Results.put("CalledMethod" +
                          counter,ResultSet.getString("CalledMeth
                          od"));
                    counter++;
              }
        }
        catch (SQLException e)
        {
              System.out.println("SQL Exception" + e);
        }
        finally
        {
              Stat.close();
              ResultSet.close();
              connection.close();
        }
        return Results;
  }
}
```

APPENDIX C

Source Code: FileHandler.java

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

public class FileHandler
{
      int bracketCounter = 0;
      String CurrentClassName = "";
      String CurrentNestedClass = "";
      String CurrentMethodName = "";
      String ExtendedClass = "";
      String NestedExtendedClass ="";
      HashMap ObjectList = new HashMap();
      HashMap DeclaredMethods = new HashMap();
      HashMap CalledMethods = new HashMap();

      public void readFile(File file) throws IOException,
SQLException
      {
            try
            {
                  FileInputStream fis = new FileInputStream(file);
                  BufferedInputStream bis = new
                        BufferedInputStream(fis);
                  DataInputStream dis = new DataInputStream(bis);
                  String sText = "";
                  StringBuffer sResult = new StringBuffer("");

                  while ((sText= dis.readLine()) != null)
                  {
                        sText = sText.replaceAll("^\\s+", "");
                        sText = sText.replaceAll("\\s+$","");
                        sText = " " + sText;

                        sResult.append(sText);
                  }
                  int index1 = 0;
                  int index2 = 0;
                  int index3 = 0;

                  StringBuffer sTemp = new StringBuffer("");
                        while (sResult.length() != 0)
                  {
                        index1 = sResult.indexOf("{");
                        index2 = sResult.indexOf(";");
```

```
                  index3 = sResult.indexOf("}");

                  if (index3 == 0)
                  {
                        sTemp.replace(0, sTemp.length(),
                              sResult.substring(0,index3+1));

                              if (sResult.length() > 2)
                                    sResult.replace(0,
                                          sResult.length(),
                                          sResult.substring(ind
                                          ex3+2,
                                          sResult.length()));
                              else
                                    sResult.replace(0,sResult.l
                                    en
                                          gth(), "");
                  }
                  else if ((index1 < index2) && (index1 != -1))
                  {
                        sTemp.replace(0, sTemp.length(),
                              sResult.substring(0, index1+1));
                        sResult.replace(0, sResult.length(),
                              sResult.substring(index1+2,
                              sResult.length()));
                  }
                  else if ((index2 < index3) && (index2 != -1))
                  {
                        sTemp.replace (0,
                              sTemp.length(),sResult.substring(
                              0, index2+1));
                        sResult.replace(0, sResult.length(),
                              sResult.substring(index2+2,
                              sResult.length()));
                  }
                  else
                  {
                        sTemp.replace(0, sTemp.length(),
                              sResult.substring(0, index3+1));
                        if (sResult.length() > 2)
                              sResult.replace(0,
                                    sResult.length(),
                                    sResult.substring(index3+2,
                                    sResult.length()));
                        else

sResult.replace(0,sResult.length(),
                                    "");
                  }

                  evaluateLine(sTemp);
            }
            //go through the CalledMethods and see if they are
                  declared in the Classes read...
            int Dcounter = DeclaredMethods.size()/2;
            int Ccounter = CalledMethods.size()/4;
```

```
String CalledClass = "";
String CalledMethod = "";
String DeclaredClass = "";
String DeclaredMethod = "";
boolean bFound = false;

for (int i = 1; i<=Ccounter; i++)
{
      bFound = false;
      CalledClass = (String)
            CalledMethods.get("Class" + i);
      CalledMethod = (String)
            CalledMethods.get("Method" + i);
      String CuMethod = (String)
            CalledMethods.get("CurrentMethod" + i);

      for (int j = 1; j<=Dcounter; j++)
      {
            DeclaredClass = (String)
                  DeclaredMethods.get("Class" + j);
            DeclaredMethod = (String)
                  DeclaredMethods.get("Method" +
            j);

            if
                  (CalledClass.equalsIgnoreCase(Dec
                  laredClass) &&
                  CalledMethod.equalsIgnoreCase(Dec
                  laredMethod))
            {
                  //save to the database- as in
                  that class
                  DatabaseMethods
                        dataMethods = new
                        DatabaseMethods();
                  if (CalledClass.length() != 0 &&
                        CuMethod.length()!=0 &&
                        CalledClass.length()!=0 &&
                        CalledMethod.length()!=0)

                        dataMethods.insertCode(Call
                        ed
                              Class.trim(),
                              CuMethod.trim(),
                              CalledClass.trim(),
                              CalledMethod.trim());

                  bFound = true;
            }
            if (bFound)
                  break;
      }
      if (!bFound)
      {
            //save to the database as inherrited
            String XClass = (String)
```

```
                    CalledMethods.get("ExtendedClass"
                    + i);
                DatabaseMethods dataMethods = new
                    DatabaseMethods();
                if (CalledClass.length() != 0 &&
                    CuMethod.length()!=0 &&
                    XClass.length()!=0 &&
                    CalledMethod.length()!=0)

                    dataMethods.insertCode(CalledClas
                    s.trim(), CuMethod.trim(),
                    XClass.trim(),
                    CalledMethod.trim());

            }
        }

        fis.close();
        bis.close();
        dis.close();
    }
    catch (IOException e)
    {
    }
}

public void evaluateLine(StringBuffer sInput) throws
SQLException
{
    Pattern pattern = null;
    Matcher matcher = null;
    Constants myConstants = new Constants();
    int inputLength = sInput.length();

    for (int i = 0; i<inputLength; i++)
    {
        if (sInput.charAt(i) == '{')
        {
            bracketCounter++;
        }
        if (sInput.charAt(i) == '}')
        {
            bracketCounter--;
        }
    }
    if (bracketCounter == 1)
    {
        CurrentNestedClass ="";
        NestedExtendedClass = "";
    }

    //set up the current class pattern
    String currentClass = myConstants.currentClass;
    pattern = Pattern.compile(currentClass);
    matcher = pattern.matcher(sInput);
    if (matcher.find())     //if i find a new class
```

```
        {
            getCurrentClassName(sInput);
            return;
        }

        //get any new class instantiation (object)
        String Objects = myConstants.Objects;
        pattern = Pattern.compile(Objects);
        matcher = pattern.matcher(sInput);
        if (matcher.find())     //if the line contains an object
        {
            getClassName(sInput);
            return;
        }
        else  //look for the other String declaration
        {
            String StringObjects = myConstants.StringObjects;
            pattern = Pattern.compile(StringObjects);
            matcher = pattern.matcher(sInput);
            if (matcher.find())     //if the line contains
                an object
            {
                getClassName(sInput);
                return;
            }
        }

        //if the bracketCounter is greater than 0- then i need to
            look for methods
        if (bracketCounter > 0)
        {
            //set up the current method declaration pattern
            String currentMethod = myConstants.currentMethod;
            pattern = Pattern.compile(currentMethod);
            matcher = pattern.matcher(sInput);
            if (matcher.find())     //if i find a new method
            {
                getCurrentMethodName(sInput);
                return;
            }
            //this needs to be done last and if it passes the
                other test before it
            //get any method calls
            String Methods = myConstants.Methods;
            pattern = Pattern.compile(Methods);
            matcher = pattern.matcher(sInput);
            if (matcher.find())
            {
                getMethodName(sInput);
                return;
            }
        }
    }

    public void getCurrentClassName(StringBuffer Line)
    {
```

```
      StringTokenizer st = new
            StringTokenizer(Line.toString());
      String Next = "";

      while (st.hasMoreTokens())
      {
            Next = st.nextToken();
            if (Next.equalsIgnoreCase("class"))
            {
                  if (bracketCounter < 2)
                        CurrentClassName = st.nextToken();
                  else
                        CurrentNestedClass = st.nextToken();
            }
            if (Next.equalsIgnoreCase("extends"))
            {
                  if (bracketCounter < 2)
                        ExtendedClass = st.nextToken();
                  else
                        NestedExtendedClass = st.nextToken();
            }
      }
}

public void getCurrentMethodName(StringBuffer Line)
{
      StringTokenizer st = new
            StringTokenizer(Line.toString());
      String Next = "";
      String Temp = "";
      String Class = "";
      //handle else if
      if (Line.toString().contains("else if"))
      {
            return;
      }
      while (st.hasMoreTokens())
      {
            Next = st.nextToken();
            if (Next.contains("("))
            {
                  int index = Next.indexOf("(");
                  if (index != 0)
                        CurrentMethodName = Next.substring(0,
                              index);
                  else
                        CurrentMethodName = Temp;
            }
            Temp = Next;
      }

      //i want to store all declared methods to a hashmap
      int counter = DeclaredMethods.size()/2;
      counter++;
      if (CurrentNestedClass.length()>0)
            Class = CurrentNestedClass;
```

```
        else
              Class = CurrentClassName;
        DeclaredMethods.put("Class" + counter, Class);
        DeclaredMethods.put("Method" + counter,
              CurrentMethodName);
}

public void getClassName(StringBuffer Line)
{
        StringTokenizer st = new
              StringTokenizer(Line.toString());
        String Class = st.nextToken();
        Class.trim();
        String Reference = st.nextToken();
        Reference.trim();
        int counter = ObjectList.size()/2;
        counter++;

        ObjectList.put("Class" + counter, Class);
        ObjectList.put("Reference" + counter, Reference);
}

public void getMethodName(StringBuffer Line) throws
SQLException
{
        String Next = "";
        String Class = "";
        String Reference = "";
        String Method = "";
        int index = 0;
        int index2 = 0;

        Next = Line.toString();
        if (Next.contains("."))
        {
              index = Next.indexOf(".");
              index2 = Next.indexOf("(");
              int index3 = Next.indexOf("=");
              int index4 = Next.indexOf(":");
              int index5 = Next.indexOf(")");

              ///check for reserve words
              if (index2 == -1)
              {
                    return;
              }
              else if ((index2< index))
              {
                    String subNext = Next.substring(0, index2);
                    if
                          (subNext.trim().contains("if")||subNext
                    .t
                          rim().contains("catch")||
                          subNext.trim().contains("do")||subNext.
                    tr
```

```
                im().contains("for")||subNext.trim().co
        nt
                ains("return")||
                subNext.trim().contains("switch")||subN
        ex
                t.trim().contains("while"))
        {
                String inside =
                        Next.substring(index2+1,
                        Next.length());
                if (inside.indexOf(".") == -1)
                {
                        return;
                }
                else
                {
                        StringBuffer sbinside = new
                                StringBuffer();
                        sbinside.append(inside);
                        getMethodName(sbinside);
                        return;
                }
        }
        else
        {
                if (subNext.length() == 0)
                {
                        return;
                }
                else
                {
                        if (index2 < index && index5 <
                                index)      //this is for
                                casting
                        {
                                String inside =
                                        Next.substring(index5
                                +1
                                        , Next.length());
                                StringBuffer sbinside = new
                                        StringBuffer();
                                sbinside.append(inside);
                                getMethodName(sbinside);
                                return;
                        }
                }
        }
}
if (index3 != -1|| index4 != -1)
{
        if (index3 != -1)
                Reference  = Next.substring(index3+1,
                        index);
        else
                Reference = Next.substring(index4+1,
                        index);
```

```
}
else
      Reference = Next.substring(0, index);
if (index2 != -1) //there is a paranthesis
      Method = Next.substring(index+1, index2);
else        //there is not a paranthesis
      Method = Next.substring(index+1);

//get the class name
int counter = ObjectList.size()/2;
String Temp = "";
for (int i = 1; i<=counter; i++)
{
      Temp = (String) ObjectList.get("Reference" +
            i);
      if (Temp.equalsIgnoreCase(Reference.trim()))
      {
            Class = (String) ObjectList.get("Class"
                  + i);
      }
}
if (Class.length()==0)  //the reference was not
      found- it must be static
{
      if
            (Reference.trim().equalsIgnoreCase("sup
            er"))
      {
            if (CurrentNestedClass.length()>0)
            {
                  Class = NestedExtendedClass;
            }
            else
            {
                  Class = ExtendedClass;
            }
      }
      else if
            (Reference.trim().equalsIgnoreCase("thi
            s"))
      {
            if (CurrentNestedClass.length()>0)
            {
                  Class = CurrentNestedClass;
            }
            else
            {
                  Class = CurrentClassName;
            }
      }
      else
      {
            Class = Reference;
      }
}
//insert into database
```

```
      if ((CurrentNestedClass.length() != 0 ||
            CurrentClassName.length() !=0) &&
            CurrentMethodName.length()!=0 &&
            Class.length()!=0 && Method.length()!=0)
      {
            DatabaseMethods dataMethods = new
                  DatabaseMethods();
            if (CurrentNestedClass.length()>0)
            {
                  dataMethods.insertCode(CurrentNestedCla
            ss
                        .trim(),
                        CurrentMethodName.trim(),
                        Class.trim(), Method.trim());
            }
            else
            {
                  dataMethods.insertCode(CurrentClassName
            .t
                  rim(), CurrentMethodName.trim(),
                  Class.trim(), Method.trim());
            }
      }
}
else if (Next.contains("("))  //there is no dot operator
{
      String ExtendedClass2 = "";
      index = Next.indexOf("(");
      int index1 = -1;
      index1 = Next.indexOf("=");
      if (index != 0)   //has a paranthesis
      {
            if ((index< index1) && index1 != -1)
                  Method = Next.substring(0, index);
            else if (index1 != -1)
                  Method = Next.substring(index1+1,
                        index);
            else
                  Method = Next.substring(0, index);
            //check for reserve words
            if
                  (Method.trim().equalsIgnoreCase("if")||
                  Method.trim().equalsIgnoreCase("catch")
                  ||
                  Method.trim().equalsIgnoreCase("do")||M
                        ethod.trim().equalsIgnoreCase("fo
                        r")||Method.trim().equalsIgnoreCa
                        se("return")||Method.trim().equal
                        sIgnoreCase("switch")||Method.tri
                        m().equalsIgnoreCase("while"))
            {
                  return;
            }
            if (CurrentNestedClass.length()>0)
            {
                  Class = CurrentNestedClass;
```

```
                    ExtendedClass2 = NestedExtendedClass;
                }
                else
                {
                    Class = CurrentClassName;
                    ExtendedClass2 = ExtendedClass;
                }
                //i want to store all called methods to a
                    hashmap
                int counter = CalledMethods.size()/4;
                counter++;

                CalledMethods.put("Class" + counter,
                    Class.trim());
                CalledMethods.put("Method" + counter,
                    Method.trim());
                CalledMethods.put("CurrentMethod" + counter,
                    CurrentMethodName.trim());
                CalledMethods.put("ExtendedClass" + counter,
                    ExtendedClass2.trim());
            }
        }
    }
}
```

APPENDIX D

Source Code: GenerateDiagrams.java

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class GenerateDiagrams extends JPanel
{
	public void generateDiagramByClass()
	{
		DatabaseMethods databaseMethods = new DatabaseMethods();
		try
		{
			//call the database to get the classes and methods
			HashMap Results =
				databaseMethods.getDiagramInfoByClass();
			int size = Results.size()/4;
			String CurrentClass = "";
			String CurrentMethod = "";
			String CalledClass = "";
			String CalledMethod = "";
			String PreviousClass = "";
			String PreviousMethod = "";
			String NextCurrentClass = "";
			String NextCurrentMethod = "";
			String NextCalledClass = "";
			String NextCalledMethod = "";
			Constants myConstants = new Constants();

			JFrame frame2 = new JFrame("Diagram By Class");
			JPanel pClass = new JPanel();
			pClass.setLayout(new BoxLayout(pClass,
				BoxLayout.Y_AXIS));
			JPanel pTitle = new JPanel();
			JLabel lTitle = new JLabel(new
				ImageIcon(myConstants.classTitle));
			lTitle.setBorder(BorderFactory.createLineBorder(Col
				or.black));
			pTitle.add(lTitle);
			pTitle.setBorder(BorderFactory.createEmptyBorder(8,
				8, 8, 8));

			//set up the frame
			frame2.setSize(800, 400);
			frame2.getContentPane().setLayout(new
				BoxLayout(frame2.getContentPane(),
```

```
      BoxLayout.Y_AXIS));
frame2.getContentPane().add(pTitle);


for (int i = 1; i<=size; i++)
{
      CurrentClass = (String)
            Results.get("CurrentClass" + i);
      JPanel pCurrentClass = new JPanel();
      if (!PreviousClass.equals(CurrentClass))
      {
            pCurrentClass.setLayout(new
                  BoxLayout(pCurrentClass,
                  BoxLayout.Y_AXIS));
            pCurrentClass.setBorder(BorderFactory.c
            re
                  ateLineBorder(Color.BLACK));
            JLabel lCurrentClass = new
                  JLabel(CurrentClass,
                  SwingConstants.LEFT);
            Font labelFont1 =
                  lCurrentClass.getFont();
            Font labelFont2 =
                  labelFont1.deriveFont(16.0f);
            lCurrentClass.setFont(labelFont2);
            pCurrentClass.add(lCurrentClass);

            PreviousMethod = "";

            for (int j=i; j<=size; j++)
            {
                  CurrentMethod = (String)
                        Results.get("CurrentMethod"
                  +
                        j);
                  if
                        (!PreviousMethod.equals(Cur
                  re
                        ntMethod))
                  {
                        JPanel pCurrentMethod = new
                              JPanel();
                        JLabel lCurrentMethod = new
                              JLabel(CurrentMethod)
                        ;
                        Font labelFont3 =
                              lCurrentMethod.getFon
                              t();
                        Font labelFont4 =
                              labelFont3.deriveFont
                              (16.0f);
                        lCurrentMethod.setFont(labe
                        lF
                              ont4);
                        pCurrentMethod.add(lCurrent
                  Me
```

```
        thod);
 JLabel arrow1 = new
        JLabel(new
        ImageIcon(myConstants
        .arrow2));
 pCurrentMethod.add(arrow1);
 CalledClass = (String)
        Results.get("CalledCl
        ass" + j);
 CalledMethod = (String)
        Results.get("CalledMe
        thod" + j);

 JPanel pCalled = new
        JPanel();
 pCalled.setLayout(new
        BoxLayout(pCalled,
 BoxLayout.Y_AXIS));
 pCalled.setBorder(BorderFac
 to
        ry.createLineBorder(C
 ol
        or.BLACK));
 JLabel lCalled = new
        JLabel(CalledClass +
        "." + CalledMethod);
 Font labelFont5 =
        lCalled.getFont();
 Font labelFont6 =
        labelFont5.deriveFont
        (16.0f);

lCalled.setFont(labelFont6);
 pCalled.add(lCalled);

 for (int k = j+1;
        k<=size; k++)
 {
        NextCurrentClass =
               (String)
               Results.get("Cu
        rr
               entClass" + k);
        NextCurrentMethod =
               (String)
               Results.get("Cu
               rrentMethod" +
               k);
        NextCalledClass =
               (String)
               Results.get("Ca
               lledClass" +
               k);
        NextCalledMethod =
               (String)
               Results.get("Ca
```

```
                    lledMethod" +
                    k);

                if
                    (NextCurrentCla
                ss
                    .equalsIgnoreCa
                se
                    (CurrentClass)
                &&
                    NextCurrentMeth
                od
                    .equalsIgnoreCa
                se
                    (CurrentMethod)
                )
                {
                    JLabel lCurrent
                        =
                        new
                        JLabel(Ne
                        xtCalledC
                        lass +
                        "." +
                        NextCalle
                        dMethod);
                    Font labelFont7
                        =
                        lCalled.g
                        et
                        Font();
                    Font labelFont8
                        =
                        labelFont
                        7.deriveF
                        ont(16.0f
                        );
                    lCurrent.setFon
                    t(labelFont8);
                    pCalled.add(lCu
                    rrent);
                    pCurrentMethod.
                    add(pCalled);
                    j++;
                }
                else
                {
                    break;
                }
            }


    pCurrentMethod.add(pCalled);
        pCurrentClass.add(pCurrentM
        ethod);
        PreviousMethod =
```

```
                            CurrentMethod;
                        PreviousClass =
                            CurrentClass;
                        if
                            (!NextCurrentClass.eq
                            uals(CurrentClass))
                            break;
                    }
                    else
                    {
                        break;
                    }
                }
                pClass.add(pCurrentClass);
            }
        }

        JScrollPane scroll = new JScrollPane(pClass);
        scroll.setVerticalScrollBarPolicy(ScrollPaneConstan
        ts.VERTICAL_SCROLLBAR_ALWAYS);
        frame2.getContentPane().add(scroll);

        //Create and set up the window.
        frame2.setDefaultCloseOperation(frame2.DISPOSE_ON_C
            LOSE);
        //Display the window.
        frame2.pack();
        frame2.setVisible(true);

    }
    catch (Exception e)
    {

    }

}

public void generateDiagramByMethod()
{
    DatabaseMethods databaseMethods = new DatabaseMethods();

    try
    {
        //call the database to get the classes and methods
        HashMap Results =
            databaseMethods.getDiagramInfoByMethod();
        int size = Results.size()/4;
        String CurrentClass = "";
        String CurrentMethod = "";
        String CalledClass = "";
        String CalledMethod = "";
        String PreviousClass = "";
        String PreviousMethod = "";
        String NextCurrentClass = "";
        String NextCurrentMethod ="";
        String NextCalledClass = "";
```

```
String NextCalledMethod = "";
Constants myConstants = new Constants();

JFrame frame2 = new JFrame("Diagram By Method");
JPanel pMethod = new JPanel();
pMethod.setLayout(new BoxLayout(pMethod,
      BoxLayout.Y_AXIS));
JPanel pTitle = new JPanel();
JLabel lTitle = new JLabel(new
      ImageIcon(myConstants.methodTitle));
lTitle.setBorder(BorderFactory.createLineBorder(Col
or
      .black));
pTitle.add(lTitle);
pTitle.setBorder(BorderFactory.createEmptyBorder(8,
      8, 8, 8));

//set up the frame
frame2.setSize(800, 400);
frame2.getContentPane().setLayout(new
      BoxLayout(frame2.getContentPane(),
      BoxLayout.Y_AXIS));
frame2.getContentPane().add(pTitle);

for (int i = 1; i<=size; i++)
{
      CalledClass = (String)
            Results.get("CalledClass" + i);

      JPanel pCalledClass = new JPanel();
      if (!PreviousClass.equals(CalledClass))
      {
            pCalledClass.setLayout(new
                  BoxLayout(pCalledClass,
                  BoxLayout.Y_AXIS));
            pCalledClass.setBorder(BorderFactory.cr
      ea
                  teLineBorder(Color.BLACK));
            JLabel lCalledClass = new
                  JLabel(CalledClass,
                  SwingConstants.LEFT);
            Font labelFont1 =
                  lCalledClass.getFont();
            Font labelFont2 =
                  labelFont1.deriveFont(16.0f);
            lCalledClass.setFont(labelFont2);
            pCalledClass.add(lCalledClass);

            for (int j=i; j<=size; j++)
            {
                  CalledMethod = (String)
                        Results.get("CalledMethod"
                  +
                        j);
                  if
```

```
         (!PreviousMethod.equals(Cal
le
        dMethod))
{
        JPanel pCalledMethod = new
                JPanel();
        JLabel lCalledMethod = new
                JLabel(CalledMethod);
        Font labelFont3 =
                lCalledMethod.getFont
        ()
                ;
        Font labelFont4 =
                labelFont3.deriveFont
        (1
                6.0f);
        lCalledMethod.setFont(label
        Font4);
        pCalledMethod.add(lCaledMet
        hod);
        JLabel arrowl = new
                JLabel(new
                ImageIcon(myConstants
        .arrow3));
        pCalledMethod.add(arrowl);
        CurrentClass = (String)
                Results.get("CurrentC
        lass" + j);
        CurrentMethod = (String)
                Results.get("CurrentM
        ethod" + j);
        JPanel pCurrent = new
        JPanel();
        pCurrent.setLayout(new
                BoxLayout(pCurrent,
                BoxLayout.Y_AXIS));
        pCurrent.setBorder(BorderFa
        ctory.createLineBorder(Co
                lor.BLACK));
        JLabel lCurrent = new
                JLabel(CurrentClass +
                "." + CurrentMethod);
        Font labelFont5 =
                lCurrent.getFont();
        Font labelFont6 =
                labelFont5.deriveFont
        (16.0f);

        lCurrent.setFont(labelFont6
        );
        pCurrent.add(lCurrent);
        for (int k = j+1; k<=size;
                k++)
        {
                NextCurrentClass =
                        (String)
```

```
        Results.get("Cu
rrentClass" + k);
NextCurrentMethod =
        (String)
        Results.get("Cu
rrentMethod" + k);
NextCalledClass =
        (String)
        Results.get("Ca
lledClass" + k);
NextCalledMethod =
        (String)
        Results.get("Ca
        lledMethod" +
        k);
NextCalledClass +
        NextCalledMetho
        d);
If
        (NextCalledClas
        s.equalsIgnoreC
        ase(CalledClass
        ) &&
        NextCalledMetho
        d.equalsIgnoreC
        ase(CalledMetho
        d))
{
        JLabel lNext =
                new
                JLabel(Ne
                xtCurrent
                Class +
                "." +
                NextCurre
                ntMethod)
                ;
        Font labelFont7
        =
                lNext.get
        Font();
        Font labelFont8
        =
                labelFont
        7.
                deriveFon
        t(
                16.0f);
        lNext.setFont(l
        abelFont8);
        pCurrent.add(lN
        ext);

        pCalledMethod.a
        dd(pCurrent);
        j++;
```

```
                                }
                                else
                                    break;
                            }

    pCalledMethod.add(pCurrent);
                            pCalledClass.add(pCalledMet
                            hod);
                            PreviousMethod =
                                CalledMethod;
                            if
                                (!NextCalledClass.equ
                            als(CalledClass))
                                    break;
                        }
                    }
                    pMethod.add(pCalledClass);
                }
                PreviousClass = CalledClass;
            }
            JScrollPane scroll = new JScrollPane(pMethod);
            scroll.setVerticalScrollBarPolicy(ScrollPaneConstan
            ts.VERTICAL_SCROLLBAR_ALWAYS);
            frame2.getContentPane().add(scroll);

            //Create and set up the window.
            frame2.setDefaultCloseOperation(frame2.DISPOSE_ON_C
            LOSE);
            //Display the window.
            frame2.pack();
            frame2.setVisible(true);
        }
        catch (Exception e)
        {
        }
    }
}
```

APPENDIX E

Source Code: MainFrame.java

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.*;

public class MainFrame extends JPanel implements ActionListener
{
      //frame
      private JFrame frame = new JFrame("Method Level Dependency
            Generator");
      //Panels
      private JPanel pTitle = new JPanel();
      private JPanel pUploaded = new JPanel();
      private JPanel pBottom = new JPanel();
      private JPanel pBottom2 = new JPanel();
      private JPanel pFileChooser = new JPanel();
      private JPanel buttonPanel = new JPanel();
      private JPanel pGenerator = new JPanel();

      //Labels
      Constants myConstants = new Constants();
      private JLabel lTitle = new JLabel(new
            ImageIcon(myConstants.title));
      private JLabel lUploaded = new JLabel("Files Uploaded:",
            SwingConstants.LEFT);
      private JLabel lgenerator = new JLabel(new
            ImageIcon(myConstants.generator));

      // Buttons
      private JButton bReset = new JButton("Reset Application");
      private JButton bOpenFile = new JButton("Open File To Read");
      private JButton bGenerateByClass = new JButton ("Generate By
            Class Dependencies");
      private JButton bGenerateByMethod = new JButton ("Generate By
            Method Dependencies");

      // Menu
      private JMenuBar m = new JMenuBar(); // Menubar
      private JMenu mFile = new JMenu("File");
      private JMenuItem miQuit = new JMenuItem("Quit");
      private JMenu mHelp = new JMenu("Help"); // Help Menu entry
      private JMenuItem miAbout = new JMenuItem("About");

      //text areas
      private JTextArea tFileList = new JTextArea();
```

```
//file chooser
private JFileChooser fc = new JFileChooser();


public MainFrame()
{
      //set up the text area
      tFileList = new JTextArea(5,20);
      tFileList.setMargin(new Insets(5,5,5,5));
      tFileList.setEditable(false);
      JScrollPane logScrollPane = new JScrollPane(tFileList);

      //Set menubar
      frame.setJMenuBar(m);

      //Build Menus
      mFile.add(miQuit);
      mHelp.add(miAbout);
      m.add(mFile);
      m.add(mHelp);

      lTitle.setVerticalAlignment(SwingConstants.TOP);
      lTitle.setBorder(BorderFactory.createLineBorder(Color.bla
            ck
            ));
      lTitle.setForeground(Color.black);
      pTitle.add(lTitle);
      pTitle.setBorder(BorderFactory.createEmptyBorder(8, 8, 8,
            8));
      lUploaded.setVerticalAlignment(SwingConstants.CENTER);
      lUploaded.setForeground(Color.black);
      Font labelFont1 = lUploaded.getFont();
      Font labelFont2 = labelFont1.deriveFont(16.0f);
      lUploaded.setFont(labelFont2);
      pUploaded.add(lUploaded);
      pUploaded.setBorder(BorderFactory.createEmptyBorder(8, 8,
            8, 8));

      //set up the the actions
      bOpenFile.addActionListener(this);
      bGenerateByClass.addActionListener(this);
      bGenerateByMethod.addActionListener(this);
      bReset.addActionListener(this);
      miQuit.addActionListener(new ListenMenuQuit());

      //Add Buttons
      buttonPanel.add(bOpenFile);
      pBottom.add(bGenerateByClass);
      pBottom.add(bGenerateByMethod);
      pBottom2.add(bReset);

      pGenerator.add(lgenerator);

      //set up the frame
      frame.setSize(800, 400);
```

```
    frame.getContentPane().setLayout(new
        BoxLayout(frame.getContentPane(),
    BoxLayout.Y_AXIS));
    frame.getContentPane().add(pTitle);
    frame.getContentPane().add(buttonPanel);
    frame.getContentPane().add(pFileChooser);
    frame.getContentPane().add(pUploaded);
    frame.getContentPane().add(logScrollPane);
    frame.getContentPane().add(pBottom);
    frame.getContentPane().add(pBottom2);
    frame.getContentPane().add(pGenerator);
    // Allows the Swing App to be closed
    frame.addWindowListener(new ListenCloseWdw());
}
public class ListenMenuQuit implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}

public class ListenCloseWdw extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
   }
}

public void actionPerformed(ActionEvent e)
{
    GenerateDiagrams generateDiagrams = new
        GenerateDiagrams();
    //Handle button action.
    if (e.getSource() == bOpenFile)
    {
        int returnVal = fc.showOpenDialog(MainFrame.this);
        if (returnVal == JFileChooser.APPROVE_OPTION)
        {
            File file = fc.getSelectedFile();
            tFileList.append(file.getName() + "\n");
            FileHandler fileHandler = new FileHandler();
            try
            {
                fileHandler.readFile(file);
            }
            catch (Exception e2)
            {
            }
        }
    }
    else if (e.getSource() == bGenerateByClass)
    {
        generateDiagrams.generateDiagramByClass();
```

```
        }
        else if (e.getSource() == bGenerateByMethod)
        {
              generateDiagrams.generateDiagramByMethod();
        }
        else if (e.getSource() == bReset)
        {
              DatabaseMethods dataMethods = new
                    DatabaseMethods();
              try
              {
                    dataMethods.resetApplication();
              }
              catch (Exception e3)
              {
              }
        }
  }

  private void ShowGUI()
  {
        //Create and set up the window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
  }

  public static void main(String[] args)
  {
        MainFrame mf = new MainFrame();
        mf.ShowGUI();
  }
}
```

VITA

Lesley Hays has a Bachelor of Science degree from the University of North Florida in Computer and Information Sciences, 2003 and expects to receiver a Master of Science in Computer and Information Sciences from the University of North Florida, December 2007. Dr. Robert Roggio of the University of North Florida is serving as Lesley's thesis advisor. Lesley is currently employed as a Software Engineer II at CACI, Inc. She has been with the company for over four years.

Lesley has on-going interests in reverse engineering software code. Lesley has programming experience in Java, Java Servlets, XML, XSL, JavaScript and has utilized Jakarta Struts. Lesley's academic work has included COBOL, C, and Visual Basic.