An implementation of refactoring techniques in the Java language

Jonathan Owen Hugh Nicholson
Registration Number: 0226904

Supervisor: David Lyons
University of Essex

Produced with LaTeX

March 2005

Abstract

This is a technical project aimed at those who wish to learn more about implementing refactoring techniques on Java source code. It does so by walking through the process of designing and building an Application Program Interface (API) and demonstration application. The system outlined here uses a symbol tree, in comparison to the more traditional symbol table, to store identifier definition information. The report gives evidence leading to the conclusion that the symbol table is the more efficient option for such a task, and therefore a list of possible extensions are included.

Acknowledgements

I would like to acknowledge the following people for all the help and contributions they have provided during the course of this project. Firstly David Lyons, my supervisor; David Rosenstrauch, for his help and advice when starting my project; Matthew Hunt, Mike Gill & Andrew Stephen, for their interesting comments and opinions during program development; My family for proof reading the report; finally Terence Parr, for his work on parser generation and more specifically for providing the grammar files I've used in this project.

Contents

List of Figures

Chapter 1

Introduction & Overview

This chapter provides an introduction to both the project and this report. It outlines the proposed goals/limitations of the project, defines what refactoring is, and gives some information on related work in the field.

It is worth pointing out at the start of this report that appendix A, page 54, contains a directory listing of the included CD's contents. It also details which files were authored by a third party.

1.1 A brief introduction to Java

One of the current most common programming languages on the Internet today is Java, but despite its now grand presence, it had much humbler beginnings. Originally conceived in 1991 as a means to an end; Java was intended to be used in communication of network enabled household devices [J.B05]. Since May 1995 when the initial version was publicly released, Java has swept through the computing industry faster than any other language. One of the very attractive features of Java is that a correctly programmed application will run the same on one machine as it does on another despite hardware/architecture and software changes, as long as a Virtual Machine exists for that platform. Because of this Java can be found all over the Internet in the form of applets or Java Server Pages (JSP), applications on your desktop, and also in Operating Systems (OS) for both embedded and desktop systems, JavaOS for example.

1.2 What is refactoring?

A simple fact of programming in any language is that no program is perfect. The chance of imperfections, or bugs, appearing in a program increases as its complexity and size increases. How a piece of software is maintained for future updates must be under constant scrutiny. A good programmer, or programming team, will work to make it easier for future developers to modify their code. One such way of achieving more maintainable code is a technique called ”refactoring”.

Martin Fowler describes refactoring as ”a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.”[M.F05].

”I certainly use IDE based refactoring all the time and feel that the need for constant grooming / refactoring of long-term projects is one of the key lessons I learned building jguru.com” - Terence Parr, Appendix ”Correspondence with Terence Parr”, page 88.

Refactoring attempts to rid code of ”Bad Smells” [M.F04][G.G05] by performing small transformations on the source code. Each transformation does little, but by performing them in sequence they can produce vastly restructured code. Its worth pointing out here that although the resulting code from a refactoring tool may be more maintainable, it may no longer meet the predefined requirements of the software. For example if there were a requirement that the code should execute within a given length of time, certain code transformations might cause it to exceed the allotted time. Another concern is that if the refactoring tool were improperly programmed it could cause errors within the code it outputs. For these reasons constant regular testing is recommended whenever refactoring techniques are employed to ensure that there is no detriment to the overall performance of the system under development.

1.3 Related Work

The earliest refactoring tool was a PhD project called "Refactoring Browser" [Unk05a] developed for the Smalltalk language [P.W04]. Smalltalk itself has been around since the 1960's, this tool was most likely created mid 1990's. Recently there have been many more applications that provide refactoring features; unsurprisingly the majority of these applications are Integrated Development Environments (IDE). For example IntelliJ, Eclipse, and NetBeans are all IDEs for the Java language that support refactoring to varying degrees. However not all applications that support refactoring techniques can be considered IDEs, such as JRefactory [M.A05] and Transmogrify[Unk05b]. Despite more applications providing refactoring features there is not one that covers every possible technique, one system may offer more than another and the choice of which application to use is greatly a matter of requirements and personal preference. For example the software for this project was developed using Eclipse as it is free and has ANTLR (discussed later) support.

Aside from commercial and freely available applications, refactoring is still an active area of research. For example Kent University has a number of research projects, open to masters students, with refactoring as a major aspect[oK05].

1.4 Project goals and limitations overview

The overall goal of this project is to research and implement a system that performs refactoring techniques on the Java language, which can be broken down into the following objectives:

1. Research refactoring techniques and related software. Perhaps the biggest objective with no definite end point. Research will continue throughout the project's life cycle.
2. Learn how to use the applications required to complete the project.
3. Produce functional and non-functional requirements based on the initial research.
4. Design the system to meet the proposed requirements.
5. Implement the system, and solve any problems encountered.
6. Test the system, comparing results to its requirements and design.
7. Produce a detailed final report summarising outcomes and concluding the knowledge gained in the process of meeting the previously stated objectives.

This is a large task for a its purpose and therefore will be scaled down to turn it into a manageable problem. For this reason limitations are imposed in the two main project areas such that:

Input Language: To create a system that supports every aspect of Java goes beyond the intended scope of this project. Therefore it is necessary to limit how much of the language is supported by focusing on a smaller subset of Java, such that:

- *Packages and Imports:*
  The amount of parsing required must be reduced to a focused area or else there will be too many variables to take into account. The most appropriate way to do this is to ignore the use of package declarations and import statements, and work purely on the code found within a single file. This ensures that there is only one defined Abstract Syntax Tree (AST) of interest, which may contain multiple class and inner class definitions.
- *Inheritance:*
  Inheritance will be ignored for the same reasons as outlined above. If there were more time this is a section of the language that would be fun to play with. The only form of inheritance supported is the use of inner classes.
- *Loops and blocks:*
  Theoretically adding support for these is a fairly easy task, but they add more complexity to the problem of scope (discussed later), and so have been ignored for now.
- *Interfaces and Abstracts:*
  Compared to dealing with entire classes adding support for these should be a simple if not trivial task, however as they are directly linked to Inheritance and so will be disregarded for now.
- *Method overloading:*
  This will be implemented.
- *Object typing:*
  For the most part this will not be implemented, however as it is required for method overloading some type checking will be implemented. For example in the Java code:
  `Integer.toString().equals(String)`
  The first method, `Integer.toString()` must be resolved to its return type (`String`). Once this is accomplished `String.equals(String)` can be resolved.

Techniques implemented: Given the allotted time-scale for this project and the research conducted it appears feasible to implement only 2 techniques, renaming variables and renaming methods. These refactorings are closely related, the latter being a more advanced variation of the former when considering method overloading.

Chapter 2

Principles of parsing

Compiler Compilers (CC), otherwise known as Parser Generators, have existed since the 1960's[J.S63]. This chapter will provide an overview discussion covering the terminology and tasks performed by the types of applications associated with parsing.

If further detail is sought please look to resources such as those found in the Bibliography.

2.1 Context-Free Grammars

”Context-free grammars are used to describe the syntactic structure of programs of a programming language. This shows how programs are composed from subprograms, or, more precisely, which elementary constructs there are and how composite constructs can be built from other constructs”[D.M01, p.270]

Context-free grammars are written in a recursive format similar to Backus Naur Form (BNF), the formal grammatical notation of languages. The format used is implementation specific to the Parser Generator, but all try to present the rules in an analogous style. Simply the rules defined in a grammar file, and any associated code therein, are used to create Parsers.

2.2 Top-Down, Bottom-Up, LL, LR & Lookahead

The majority of the following definitions are quotations from the stated citations, in an attempt to avoid redefining the wheel.

Top-Down Parsing: ”Begin with the start symbol and apply the grammar rules forward until the symbols at the terminals of the tree correspond to the components of the sentence being parsed”
[K.K91, p.388]

Bottom-Up Parsing: ”Begin with the sentence to be parsed and apply the grammar rules backward until a single tree whose terminals are the words of the sentence and whose top node is the start symbol has been produced”
[K.K91, p.388]

Lookahead: ”When parsing a stream of input symbols, a parser has matched (and no longer needs to consider) a portion of the stream to the left of its read pointer. The next k symbols to the right of the read pointer are considered the fixed lookahead. This information is used to direct the parser to the next state.”
[T.P05b]

LALR: ”A special type of LR grammar that uses a lookahead technique to decide how to build intermediate tables when constructing a parser from a grammar specification. LALR grammar means a lookahead LR grammar. Parsers generated from formal grammar tables, like yacc, are usually bottom-up LALR parsers.”
[T.S99, Part 3 - glossary]

LL: ”An LL parser parses input from left to right using leftmost derivation. A leftmost derivation replaces the leftmost nonterminal symbol with the matching definition of a grammatical rule. (The definition for a rule or production is often

called the right-hand side or RHS for the rule. That's because the definition of the rule appears on the right while the name identifying the rule appears on the left). This replacement of nonterminal symbols by definitions occurs in a sequence of steps until all nonterminal symbols are replaced by terminal symbols. At this point the input sentence is parsed, and the recognizer can decide if the input is syntactically correct. Hand-built recursive descent parsers are LL parsers."
[T.S99, Part 3 - glossary]

LL(k): "An LL(k) parser is identical to an LL parser but needs to look ahead a certain number of tokens (symbols) before deciding which rule to be applied from a given grammar. Typically k is equal to 1 implying that one-symbol lookahead is required. Such a grammar is an LL(1) grammar. ANTLR generates LL(k) parsers from LL(k) grammars."
[T.S99, Part 3 - glossary]

LR: "An LR parser scans its input from left to right using rmost [right most] derivation. A rightmost derivation usually parsers button up instead of top down as an LL parser does. Starting with terminal symbols representing the input sentence (the opposite of an LL parser which starts with a nonterminal symbol), an LR parser matches definitions (the RHS of a rule) and replaces them with the nonterminal symbol naming the rule (the LHS of the rule)."
[T.S99, Part 3 - glossary]

LR(k): "An LR(k) parser is identical to an LR parser but needs to look ahead a certain number of tokens (symbols) before deciding which rule to be applied from a given grammar. Typically k is equal to 1 implying that one-symbol lookahead is required. Such a grammar is an LR(1) grammar."
[T.S99, Part 3 - glossary]

Related information:[R.Q96]

2.3 Parser generators

A Parser generator is a software application that takes a grammar file as input and compiles it into some sort of parser software in a given language. Exactly what it will produce will be implementation specific, for example some generators allow addition of user code within rules giving greater flexibility.

2.4 Parsers

There are three main parser types, Lexers, Recognizers, and Tree Parsers. They provide lexical, syntactic, and semantic analysis respectively. Rather than explaining the differ-

ences between parsers, differences between the tasks they perform have been presented instead.

2.4.1 Lexical analysis (Lexer)

Lexical analysers (Lexers) reads a series of characters from an input source and converts them into lexical units (symbols). To use an analogy symbols are analogous to words in a parsed document. Some symbols are discarded at this point providing a certain amount of filtering to the input stream. For example white space within source code could be ignored.

2.4.2 Syntactic analysis (Recognizer)

Sequences of valid symbols generated by lexical analysis are used to build syntactic units in the structure provided by the languages grammar. If the Recognizer finds no matching grammatical rule for a given sequence of symbols the input file is considered to have have incorrect syntax usage and parsing will stop. Therefore, to continue with the previous analogy, valid syntactic rules could be considered to be sentences, paragraphs, and finally the complete document. The analogy breaks down a little at this point as the word-sentence-paragraph-document structure is a very limited view of the recursive nature of the grammar, and therefore a syntactic unit.

As the syntactic units generated here follow a very strict structure it is possible for the recognizer to produce an AST of this data so that it can be analysed further. An AST might look like figure 2.1 below, it shows a section of the structure of a possible AST, and illustrates the recursive relationships between syntactic units.

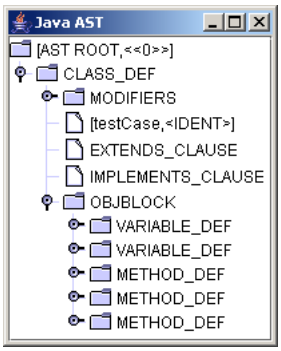


Figure 2.1: An example of an AST generated in ANTLR

2.4.3 Semantic analysis (Tree Parser)

Semantic analysis compares the result of syntactic analysis against context information. To continue with the previous analogy; a sentence may have the correct syntax, but

it may not make sense in the paragraph to which it belongs. For example the Java programming language has context information such as strict scope and visibility rules, i.e. where and how an identifier is defined will restrict its use within a given system.

As semantic analysis by definition is context checking, it cannot be performed with a context-free grammar on its own. Extra code has to be placed within the grammar file and provide this information. This is probably the most complicated of the three types of analysis, and the hardest to debug.

2.5 Conclusions

This investigation has lead to the conclusion that a Top-Down approach would best suit the purposes of this system, and therefore an LL(1), or LL(k), based parser generator is required. Which program to use will be discussed in the next chapter.

Chapter 3

JavaCC versus ANTLR

This chapter discusses two popular parser generators and the ultimate decision of which to use. To reach a conclusion there are relevant points to consider about each of these applications. These include:

Documentation: There must be adequate documentation for the beginner, and perhaps tutorials produced by a third party.

Ease of learning: It must be relatively easy and quick to start using the program, otherwise most of the available time will be spent doing this and not progressing on the central theme of the project.

Fully featured: The program must have enough features to achieve the goals of this projects. In particular it must be able to produce an AST. Extraneous features may provide future extendibility and should be considered, but these are weighted less compared to those that are essential.

3.1 JavaCC (and JTree)

JavaCC[S.V04] is a widely used Top-Down LL(1), with optional LL(k) functionality, based parser generator for the Java language. Early research indicated that JavaCC is a popular and recommended choice among enthusiasts. This is illustrated by the fact that at the time of writing a Java1.5 grammar file has recently been released, and in the JavaCC grammar repository there are over 50 grammar files.

However, it is worth noting that the documentation that is available to the beginner is minimal. Articles supposedly written for the layman require large leaps in knowledge and understanding, with very little initial visible result. An example of this is that an attempted compilation of the Java1.4 grammar produced over one hundred errors, for which no explanation could be found.

JavaCC on its own does not produce an AST, but rather this is accomplished with the use of a modified grammar and a secondary program called JTree. JTree is designed to make heterogeneous ASTs, simply meaning that the AST is constructed from different types of objects, in contrast to a homogeneous AST which is constructed from one type of object, usually with a constant to indicate its type. The differences between the two can be summarised as follows: homogeneous trees are ”simpler”, and allow traversal using a context-free grammar[T.P05c]. In the case of this project either type of tree could be employed.

3.2 ANTLR

ANTLR[T.P05a] is another widely used Top-Down LL(k) based parser generator application, boasting approximately five thousand software downloads a month. Notable features are that it has the ability to compile grammars into multiple target languages, not only Java, and that it is a single application rather than two as in the case of JavaCC (above). One small disadvantage of ANTLR is that the library file must be present in the `classpath` of the system in which it is being executed in. However this is such a trivial matter that it becomes an irrelevance.

ANTLR generates parsers that create homogeneous trees by default, but also allows generation of heterogeneous trees at the user's discretion. Although this feature provides much greater functionality, within the scope of this project it is not required.

Support for ANTLR both, official and unofficial, is widely available. Terence Parr provides a lot of resources on JGuru[T.P05c], at Sun Microsystems[T.S99], and on his own site[T.P05a].

ANTLR also provides a visual representation of the generated AST in the form of a `JTree` (not to be confused with the JTree application) displayed within a `JFrame`. This feature helps a great deal in understanding how the grammars work, debugging the system, and makes it a much more user friendly application for the beginner.

It is worth mentioning here that one of the programs listed in under the section ”Related Work” (P.3) in the Introduction, Transmogrify[Unk05b], uses ANTLR and is therefore proof that it is a viable option.

3.3 Conclusions

From the research carried out and outlined above it has been concluded that ANTLR is the most reasonable choice for this project. The ANTLR community has lots of documentation and tutorials ranging from beginner to expert, thus making it is easier to get started with ANTLR than with JavaCC.

There is one minor disadvantage to using ANTLR, which is that the most recent Java grammar supports only version 1.3, where as Java1.5 is available with JavaCC. However in the context of the project, this point is of minor importance. In the time available, and within the limitations imposed in the system, the extra features of Java1.5 would remain unused and would only add extra complexity, both from the grammar itself, and from using JavaCC instead of ANTLR.

Chapter 4

Learning ANTLR

This chapter will provides a general overview to the ANTLR grammar file format, and its associated rules. Firstly the the file structure itself will be examined, followed by a set of grammatical rules with descriptions and explanations. Finally the chapter will conclude with the creation of an expression parser written for ANTLR.

4.1 A closer look at grammar files

4.1.1 *.g file layout

This section will look at the grammar files used in ANTLR, specifically those compiled into Java programs. A general overview of the file format is provided in figure 4.1. Tree Parsers fit a very similar pattern, what differences there are will be discussed when each section of the file is described in a little more detail.

```
header
{
        // Header
}

// Class definition (Recognizer)

options
{
        // Options
}

{
        // User defined code
}

// Syntactical rules

// Class definition (Lexer)

// Lexical rules
```

Figure 4.1: Overview of the ANTLR grammar file format

- Header
  Java code that needs to be placed at the top of the generated files, before the class declaration, goes into this section. This might include a package definition and/or import statements.

- Class definitions
  A statement in the format `class x extends parser` where x is the name of the class/file to be generated. The only difference between a definition for a Recognizer and a Lexer is the class that it extends, Recognizers should extend `Parser`, and Lexers should extend `Lexer`.
  Tree Parser grammar files only have one class definition and it should extend the `TreeParser` class.

- Options
  Any options needed for the class should go here, such as lookahead and importing/exporting the vocabulary to a file. Exporting the vocabulary of the grammar produces a Java interface and a text file with a complete listing of node types, specified as integer constants. The text file is used by the `importVocab` option in other grammar files, and the interface can be used by Java applications written to examine the AST.
  More information about these options are on the ANTLR website[T.P05a].

- User defined code
  This includes any variable or method definitions that the user may wish to add into the body of the class.

- Grammatical Rules
  Both lexical and syntactic rules follow the same format, which is not too dissimilar to BNF. These rules are explained in detail in the next subsection.
  Grammatical rules for Tree Parsers are in exactly the same format as lexical or syntactic rules.

4.1.2 Grammatical rules

The general format of an ANTLR grammatical rule is as shown in figure 4.2. It shows the declaration of a rule, in this case named ”rule1”, that can either be a value or another rule, ”rule2”. This is a simplistic rule without any operator (special) characters, which will be outlined later.

Rules such as this produce methods within the class it is associated, in this case the method would be called ”rule1”. Inside this method will be a try/catch block containing a switch statement for each of the conditions that could cause this rule to be used, in this example the switch statement will contain cases for ”rule2” and ”value”. Each case statement may itself have a switch statement within it. This knowledge is useful during the debugging process, but any necessary changes to the code should be placed into the grammar. This promotes consistency and prevents changes being lost the next time the code is regenerated.

```
rule1    : rule2
         | value;
```

Figure 4.2: Example format of ANTLR grammatical rules

Further understanding of the rule format is best described through a simple example. The following figures illustrate the grammar needed to define a Lexer, Recognizer, and Tree Parser designed to solve a simple mathematical addition.

First is the grammar needed for the Lexer, as outlined in figure 4.3. It defines a number as one or more digits, denoted by the special character "+", followed by the possibility of more digits indicated by the special character "?". For the purposes of this report the term "sub-rule" will be used to identify a rule used within another. For example DIGIT is a sub-rule to NUMBER.

Notice the rule names are in upper-case, the ANTLR naming convention specifies that Lexer rules should be in upper-case only. These rules become node types for use in Recognizers and Tree Parsers, and are exported as the grammar's vocabulary.

Note: If any other characters than the ones defined are found in the input, the Lexer will complain that the input is invalid and stops parsing, this is lexical analysis. This includes white space which will be mentioned later.

```
NUMBER  : (DIGIT)+ ('.' (DIGIT)+)?;

DIGIT   : '0'..'9';

PLUS    : '+';
```

Figure 4.3: An example ANTLR grammatical rule for a Lexer

The Recognizer can then take the series of symbols generated by the Lexer and apply it to its rules, stated in figure 4.4. It defines an expression (expr) to be a NUMBER, followed by an addition (PLUS), followed by another NUMBER. Notice that lower-case is now used for rule definition, again this is the ANTLR naming convention. The "^" is a special character indicating that the preceding symbol should become the root of the syntax tree being generated. To phrase it another way PLUS is the parent and the two NUMBER type nodes will be its children.

Note: If a series of symbols generated from the Lexer do not match any rules in the Recognizer, it is considered to have incorrect syntax and parsing stops, this is syntactic analysis.

```
expr    : NUMBER PLUS^ NUMBER;
```

Figure 4.4: An example ANTLR grammatical rule for a Recognizer

Tree Parsers's rules are written in a format that has been described as analogous to LISP notation, as indicated in figure 4.5. For example the "#" character denotes which node type it will find as the root of the syntax tree generated by the Recogniser. This is the first example where Java code has been inserted on a rule; the code is executed whenever that rule is matched. These code sections can be added to almost any part of a rule by surrounding it with "{}" braces. The most significant points to place these would be before the initial ":" character, where any general declarations can be

placed. After a particular sub-rule, as in figure 4.5, the code is executed when that specific type of rule has been matched. Finally code can be inserted within the rule itself, for example in a rule such as NUMBER; this is demonstrated in figure 4.3 where there is an unknown number of components to the rule. In that case a piece of code could be inserted to, say, add the components to a `Vector`.

Also note the format of `left:NUMBER`, this allows the node of that number to be used within the associated code. Similarly a statement such as `left=NUMBER` could be used if the NUMBER rule returned a value, in this case "left" would have to be defined at the start of the rule block. It is possible to combine these two statements, such as `left=node:NUMBER`, to get both the node position and the value returned. Figure 4.6 gives an alternative if overly complex version to the previous NUMBER rule, which returns its value as a float. Where the logic resides can be essential, but for the most part it is up to the programmer. Notice the new `returns [variable definition]` line to the rule, and how everything must be defined even if it is not used in every sub-rule. This is because the structure of the generated code uses try/catch blocks, so it is possible for scoping issues to arise if variables are not defined and instantiated at the start of the method.

```
expr
        : #(PLUS left:NUMBER right:NUMBER)
        {
         float leftExpr, rightExpr;
         leftExpr = Float.parseFloat(left.getText());
         rightExpr = Float.parseFloat(right.getText());
         System.out.println(leftExpr + rightExpr);
        };
```

Figure 4.5: An example ANTLR grammatical rule for a TreeParser

```
NUMBER returns [float f = 0]
        {
         Vector l = new Vector();
         Vector r = new Vector();
        }
        : (lnode:DIGIT
                        {
                         l.add(lnode.getText());
                        }
                )+ ('.' (rnode:DIGIT
                        {
                         r.add(rnode.getText());
                        }
                )+)?
        {
         String st = "";
         if(!l.isEmpty())
          for(int i=0;i<l.size();i++)
           st+=(String)l.get(i);
         if(!l.isEmpty())
         {
          st+=".";
          for(int j=0;j<r.size();j++)
           st+=(String)r.get(j);
         }
         f = Float.parseFloat(st);
        };
```

Figure 4.6: An alternative version of the NUMBER rule (untested)

As indicated earlier in the report, if something is in the input file which cannot be matched, parsing that file will fail. In the case of a Recognizer that would indicate syntax errors and correctly stop parsing the file. However in the case of a Lexer an unrecognised character might perhaps need to be be ignored rather than passed on to cause Recognizer errors. This can be done by skipping the symbol that would otherwise be created as shown in figure 4.7, removing white space.

```
WS      : (' ' | '\t' | '\n' | '\r')
        { _ttype = Token.SKIP; };
```

Figure 4.7: Handling whitespace

4.2 Simple expression parser example

The following is an extended version of the previous examples, with complete code listings found in the appendices.

4.2.1 Goals and limitations

The goals of these grammar files, and associated main application, are as follows:

1. Create a grammar, or set of grammars, that will parse and solve an expression within a file.
2. Allow use of +, −, ⋆ and \ operators. And denote precedence by using () brackets. No other operators are supported.
3. Support single character variables, 'a' through 'z' in upper or lower case, such that they need only be defined once in the file and their value is retained for future reference.
4. If a variable is defined more than once, only store the first one encountered and discard the rest.
5. If a variable is used in the equation but not defined, request that the user enter a value that will be stored for future reference. The user will be asked only once per undefined variable.
6. Produce the result of the solved equation to the screen.
7. white space should be ignored.
8. Illegal characters and undefined symbols should cause parsing to fail.
9. The main application should demonstrate the ability to solve the given equation and alter a copy of the tree to substitute each variable for the value it contains.
10. The input file will have the format found in figure 4.8.

```
variable = value;
= expression
```

Figure 4.8: Expression parser's input file format

4.2.2 Implementation & Conclusions

Complete code listings can be found in Appendix B on page 56.

The purpose of this exercise is to learn about the interaction of rules defined within an ANTLR grammar, listed are a few points which are worth raising:

non-determinism: This was a fairly common problem that occurred. Simply it means that there are two or more possible tree structures that could be created for a given set of symbols, and the parser in question cannot decide which it should construct. However this can usually solved by increasing the lookahead in the options section, for example `k = 3;`. If this change does not correct the problem then the rules need to be checked and revised as there must be a more serious problem in the structure of the rules.

Symbol table: The expression solver uses a basic singularly scoped symbol table (discussed later) by utilizing a `Hashtable` object to store variable declarations and their associated values. When a variable is declared it is added to the symbol table, and then retrieved when it is used within the expression to be solved. As there are no issues of scope or identifier overloading this solution works extremely well. There is an extension that would be worth mentioning however, which is the ability to define sub equations rather than just numeric values for a variable. One solution to this is to modify the symbol table to store an AST. Then before the final expression is solved these ”branch” ASTs can be added to the tree in the appropriate positions. These branches could also use variables, which might themselves be branches, therefore multiple passes may have to be made to produce a completed and solvable AST.

Copying the AST: In the main application two windows are displayed, one a modified version of the other. To accomplish this without altering both trees the original tree had to be duplicated using the `ASTFactory.dupTree(AST)` method. The duplicated tree can then be freely manipulated without modifying the original.

Other than these issues the program created works, and fits the requirements listed on page 19.

It is worth mentioning here that this application is not the focus of this report, it is simply an example and demonstration of ANTLR’s capabilities. Therefore it has not undergone rigorous testing or documentation.

Chapter 5

Specification

This chapter contains the complete specification for the system. A system specification, otherwise known as a requirements definition, is defined by Sommerville as:

> ”The system’s services, constraints and goals are established with consultation with system users. They are then defined in detail and serve as a system specification.”[I.S01]

Therefore the first thing that must be identified are the users of the system. Realistically the software will not have enough features to make it a viable tool, however it is useful to those who wish to learn more about the techniques that have been employed, or with to evolve the system further. Therefore possible users are:

1. Researchers: Or more specifically, students. Those wanting to learn more about refactoring or ANTLR may use the software and read this report.
2. Programmers: Those who wish to extend the system further.

These two types of users will have fairly similar requirements and so will be treated as a single user.

5.1 Functional requirements

Functional requirements are those used to specify technical issues about the proposed software. Therefore the Functional requirements for this project are as follows.

High priority:

1. Java version: A stable and widely used version of Java should be selected. For example Java1.1 is outdated and should not be used, where Java1.5 is most recent but in less widespread circulation.

2. Relevant naming: Method names must be relevant to their purpose and not overly obscure, and Class names must describe the overall function of the class on some level. This will make it easier for a user to understand how the system works and how to use it.

3. Protection: The system must be private or protected where possible, only public where necessary.

4. Source input: Only a single file at this time, a future extension could be to increase this number.

5. Error handling: Any errors should be handled correctly, and if required reported to the user.

6. Graphical User Interface (GUI): There must be a GUI for the system to allow the users to easily make use of it.

7. Display: The GUI should have some method of viewing both the original and modified source code files, as well as the ability to display the AST.

8. Simple: It should be easy for the user to perform the implemented refactoring techniques from within the GUI.

9. Indication: The GUI should let the user know when a file is being parsed, for example a modal dialog or JProgressBar could be used.

10. Source Safety: The output of the program should not intentionally overwrite the input file, therefore output files should be placed in a separate location to the input files.

11. API: Where possible the refactoring logic should be kept within a separate package to the GUI, and should be titled ”refactoryAPI”.

12. Refactoring: At least 2 refactoring techniques should be implemented, renaming of variables and renaming of methods as discussed in the project goals and limitations section (p.3).

13. Correctness: Refactoring performed should not introduce errors as long as the input source complies to the imposed limitations (p.3)

5.2 Non-Functional requirements

Non-Functional requirements are those used to specify non-technical, or extraneous, issues about the proposed software, such as documentation or time limits. Therefore the Non-Functional requirements for this project are as follows.

High priority:

1. Documentation: It must be adequate, accurate, and understandable. It will include Javadocs. This report is a part of this documentation, and should be as insightful as possible.
2. Online Help: The system's GUI will have some form of help describing how to use it.
3. Parsing speed: Parsing source files should be under a minute, avoid infinitely recursive loops, and not cause error such as stack overflows.
4. Stability: The system will be as stable as possible, bearing in mind that input code outside the scope of the limitations of the system may cause problems.
5. Deadline: The project in its entirety must be completed by the 24th of March 2005.

Optional:

1. Distribution: Software should be able to be produced as a `*.jar` file for easy distribution.

5.3 Software Selection

The software required for this project can be placed into 4 categories, programming language, development environment, parser generator and a word processor. Decisions for the proposed software should reflect the system specification, while also matching a few requirements of its own. In this case these would be:

Familiarity: As much as possible familiar software should be used to allow full focus on the rest of the project.

Expense: Free software, or software available through the University laboratory machines should be used where appropriate.

Platform: WindowsXP will be used for project development and therefore Windows specific, or cross-platform software is favoured.

In accordance with these requirements the following software has been selected:

Language: Java1.4.2, a version of Java that is most familiar and in widespread use.

Parser generator: ANTLR, as opposed to the alternatives (see P.10).

IDE: Eclipse, free, stable, provides features such as refactoring, GUI builders and ANTLR support.

Word processor: LaTeX will be used to write this report for its compact size and Portable Document Format (PDF) file generation ability, as opposed to the more familiar Microsoft Word.

Chapter 6

Design

This chapter discusses the issues of system design based on the requirements stated in the previous chapter. It has been split into two sections, namely refactoryAPI and user application. To ensure that there is no plagiarism in any part of this project, acknowledgements must be given to work accomplished by other people. A complete list of files written by other people is included in the Appendices on page 54.

The main focus of this project is on the implementation of refactoring techniques in Java, and therefore focus will remain on the refactoryAPI. Although also an essential aspect of the system, the user application, or demonstration interface, is a secondary priority in comparison.

6.1 RefactoryAPI

6.1.1 Code logic overview

As seen in earlier chapters, parsing can be broken into three categories:

1. Lexical analysis (Lexer)
2. Syntactic analysis (Recogniser)
3. Semantic analysis (an extended Tree Parser)

ANTLR includes example grammars to perform the lexical and syntactic analysis, which in the interests of separating different authors' code will remain unmodified. Also included in the distribution is a blank Tree Parser that can be modified to perform tasks such as semantic analysis. Refactoring could be considered akin to semantic analysis so the Tree Parser is a logical start point.

In the early stages of defining the project email correspondence was established with David Rosenstrauch, who very kindly gave a brief description of the stages that need to be considered before implementing a refactoring program. What follows is a summary of the issues raised in his first email:

1. Parse the source code.
2. Complete semantic analysis, ensuring that the code is compilable.
3. Symbol table generation and resolution of references. This is referenced in the next list as "SymTab".
4. Refactor.

Each of these points had to be evaluated before an adequate design solution could be decided on, with the exception of item 1 which is dealt with by the Java Lexer and Recogniser found in the ANTLR distribution. Items 2, 3, and 4 however require more in-depth discussion as follows:

Semantics: This requires checking that every rule in the Java Language is not broken by the input code. An example of such a rule is scoping, ensuring that each identifier is used in the correct place in comparison to where it was declared. Although this section is very important, it can be negated by the use of Javac. Javac is the program supplied with the Java Standard Development Kit (SDK) (otherwise known as the Java Development Kit (JDK)) that performs compilation from source to Java Virtual Machine (JVM) byte code. This performs semantic analysis, and if it fails the output streams can be captured so that errors can be reported to the user.

SymTab: This is the most interesting task and requires the most research. Specifically it can be separated into two tasks, symbol table generation and symbol resolution as seen in the later sub sections.

Refactoring: This is probably one of the simpler tasks, provided that an adequate symbol table is generated with accurate references. This is at least true for the rename refactoring where a modification to the symbol table is needed. However more advanced techniques, such as extraction of methods, require more than modifying text in the AST. They also require the augmentation of the tree's structure. The goal of the project is to get the renaming technique working and more advanced transformations will not be discussed any more until the Conclusions chapter on page 42.

Symbol table generation

Symbol tables contain "the associated declarative information ... for every defining occurrence of an identifier"[J.S63, p.393], a simple version of which has been seen in the chapter Learning ANTLR (p.13). Identifiers are the names given to things like methods and variables, for example `Vector` is a class identifier. Assuming that the `Vector` class has not been overridden then using the full package name, `java.util.Vector`, identifies the same object. Symbol tables for an Object Oriented (OO) language advance those previously seen to include scope, where the visibility of identifiers can vary depending on where it is defined and its modifiers[J.G00].

Usually symbol tables are generated using stacks so that as a new block of code, such as a method declaration, is found, a new layer is pushed onto the stack. Declarative information is then stored in that layer until the end of the block is discovered, at which point the layer is popped from the stack and its contents written to the symbol table data structure. Once a symbol has been added to the table, its unique identifier is usually used to replace its original one in the AST; this act ensures that declarations with the same name can be treated differently to each other. In Java this is best described when considering method overloading, particularly in the case of constructors where the class also shares the same name.

Once the table is complete it can be used when performing operations on the AST, such as type checking or reference resolution.

Reference resolution

Reference resolution ensures that any non-declarative identifier used within the source code, or rather its related AST, becomes associated with the correct entry in the symbol table. This can be a tricky task at best as it is not always as simple as a name lookup. As in the example above, Java allows for many blocks within the same object to share the

same name as long as they conform to name and scope rules. If an incorrect reference is made refactoring becomes an impossibility, the transformations would inherit these errors and produce an incorrect AST. When the AST is later emitted back into source code it would either no longer be compilable, or if it does compile then it would have erroneous behaviour. Therefore when the referencing process starts for an identifier it must be sure to associate it with the correct entry in the symbol table.

Multi-part names add further complexity to the resolution process, examples of these are `Object.toString()` or the more complex concatenation of methods `Object.toString().equals(String)` for example. Cases such as these need to be considered carefully, both in the method of storage for easy retrieval, and how to then find correct entries in the symbol table.

6.1.2 Conclusions

According to the information covered so far the initial design of the refactoryAPI, where the parsing logic resides, shall be as follows:

- Development will use the iterative approach. Specifically in five main programming stages not including the testing phase.

    1. Development will begin with building a symbol table with a textual interface used to display all definitions found therein. A pre-parse Javac check will be implemented here so that any compilation errors can be used to explain erroneous results.
    2. This is followed by the construction of a GUI to the API to ease later development. The issue of how to display the symbol table is a relatively easy one: a Swing component such as a `JList` or `JTable` can be used. This will allow individual declarations within the table to be selected an modified at the will of the user. The actual design of the GUI is discussed later.
    3. Resolution of variable identifiers and the refactoring techniques can then be developed in parallel, as it is otherwise very difficult to ensure that the references created are resolved to the correct identifier.
    4. Resolution of methods can then be added to the system providing features as stated in the requirements.
    5. The code can finally be cleaned up and where possible made more efficient.

- The symbol table will store the location of the definition's sub-tree in the AST, along with a set of references for each entry stored as a `Vector`.

- During symbol table generation a set of references that are used within each block will be stored so that the resolution process needs only to work from the table, rather than the AST.
- There will be a main class to which implementing applications, such as the demonstration GUI outlined later, should use to interface with the logic.
- Exception classes will be defined so that errors can be handled correctly within any given implementing GUI. Exceptions for file problems, Javac check, and parse errors at the bare minimum will be written. In particular as the errors produced by Javac are not one line, and as such will not provide a particularly user friendly message, this exception will store the output in private variables and contain methods to retrieve it.

6.2 User Application

Designing a pretty GUI for this system is not a priority, however it must have enough functionality to demonstrate the underlying refactoring logic as designed above while being as user friendly as possible. For these reasons the following design points have been described, firstly the essential features, after which any non-essential features will be listed.

- The GUI will have menus similar in style to those found in most Windows applications. To go into further detail, there will be a file menu in which commands to open/close a source file, and to close the program. A view menu should be present which will have the option to load the AST display frame provided by ANTLR, which will be disabled if no source file has been loaded. Finally a help menu should be present to give users access to information about the system and further help.
- The GUI will provide a method to indicate to the user that it is parsing a file, while not locking up the GUI itself. This will be done using a `JProgressBar` in an indeterminate state.
- There will be clear on-screen information describing what the system is doing, or has done. This will also be logged to a file so that more detailed information can be presented if it is required. This is especially required as some operations may change the status of the GUI too quickly for the user to be able to see the text that appears before it is replaced. One further technical note is that the log file stream should be flushed after each change, this is so that if a critical error occurs and the program crashes the log file will contain status messages up to that point and hopefully give some indication of what went wrong.

- Help documentation detailing how to use the system will be included and easy to find, giving an overview of how to use the the GUI. Javadoc help files will also be accessible through the help system.
- The help system will be an intuitive interface to ease navigation. Therefore it will be implemented as a rudimentary web browser to display HTML pages (Javadocs) with hyper-linking enabled.
- There will be a a text component used to display the content of the input and output source files, and a `JComboBox` to switch between them.

Here is a list of non-essential design points that will be implemented if there is enough time:

- A configuration option shall be present allowing the user to specify options such as enable/disable logging, to set paths to log file, default input directory and output directory.

Chapter 7

Implementation

This chapter provides a discussion on the various problems encountered, and their solutions, during the implementation of the system.

7.1 RefactoryAPI

7.1.1 File name checking

The first task was to load a file into the system and confirm that it is actually a Java file. This was not a particularly hard task as it was implemented as a large `if()` statement checking that the file name is correct. For example it would ensure there are more than 5 characters long, if it were 5 or less characters long then the file name would not have the correct extension. The extension itself is then confirmed by checking the last 4 characters in the file name.

7.1.2 Javac

After this came the task of checking that the file was compilable with Javac. Originally the Javac class was to be used, but after some research on the matter it appears that this class is depreciated in Java1.5. In the interests of increasing the program's lifespan the `runtime.exec()` method was adopted. This involves letting Java make a system call the the Javac executable and roughly capturing its output. The downside to this is that it relies on another application rather than an object, and although the assumption can be made that the intended users of this application will have a Java compiler, there is no guarantee that they are using this particular software. Therefore this is a trade off between reduced system compatibility and project simplicity.

7.1.3 Symbol table, or symbol tree?

One of the main problems encountered was generating the symbol table. Unfortunately in the early stages an implementation similar to that described in the book "Compiler Design"[D.M01] could not be accomplished. However an alternative style of implementation that achieves similar results was created, namely a symbol tree.

This is best explained by using an example. When ANTLR Tree Parsers operate, an initial rule is called, or more specifically the method it has generated. For this starting rule to be true its predicate rules, or sub-rules as described earlier, must be satisfied. Therefore if a simple class is defined as in figure 7.1, and each relevant rule were to print an identifying line of text to the screen, the rules would seem to be fired in an inverse order. In this case the order is variable definition, method definition and finally class definition.

This is to be expected, and by using stacks to generate the symbol table this order can be inverted back into the order they are discovered in the AST. However at the outset of the project, symbol table generation was unfamiliar, as was the use of parser technology; so the leap between theory and practice seemed astounding. By studying the interaction between grammatical rules, an alternative idea was born.

```
public class example
{
        public static void main(String[] args)
        {
                int i = 1;
        }
}
```

Figure 7.1: An example class explaining problems in symbol table generation

Rather than using the more logical stack approach to directly output the table, a more intermediate tree structure could be accomplished. It works by modifying each grammatical rule necessary to return certain information, such as node references. This information can then be inserted into relevant wrapper classes and pushed further up the AST until it reaches the initial, or root, rule.

At this point the symbol tree will contain a set of wrapper classes, each consisting of a reference to its declarative information, and (where applicable) a set of more wrapper classes and a set of unresolved references for that block. Take for example the wrapper class called `Scope`, named to avoid confusion, which contains information about its location, unresolved references, and declarative information which could include classes, methods and variables. Each of these are modelled as `Scope`, `Method`, and `Definition` respectively, and their UML diagrams can be found in the appendix on page 70. The relationship between each of these objects is a lose representation of what they were created to represent, but they function adequately for the purposes of this project. They do not use the more efficient child-sibling approach, but rather a `Vector` of child nodes.

Once the symbol tree is constructed, each node stored within it needs to know who its parent is and where it can find the tree's root, as this information is required to be able to walk the tree later to resolve references. The `generateParents()` method performs this task by recursively populating the data. Once this has been achieved the unresolved references can begin (discussed in the next section).

Advantages

The symbol tree structure is a further abstraction from the generated AST, and therefore could be considered fairly intuitive. However perhaps its one real advantage that each node can, and does, implement the `TreeNode` interface. This makes it easier for the symbol tree to be displayed within the GUI.

It is worth noting that there is a possibility of flattening the symbol tree into a table format, however this would be an extension and has not been pursued in the course of this project.

Disadvantages

A symbol tree is not the most efficient way to approach the matter of grouping declarative information. Symbol tables are used as standard for good reason, such as:

- Table lookup times are faster, using any one of the multitude of search algorithms available, compared to the lookup times on trees. Tables can also be examined using an iterative approach rather than a recursive one inherent to a tree structure.
- Creation time is faster as there is only one set of declarations to search through, rather than in the tree model where there is one per level of the tree. In a simple package with one class with no sub-classes, there could be up to 2 layers to search through, and as the complexity of the program increases so does the amount of layers that need to be inspected.
- Tables are a more efficient at storing data compared to the tree created in this project, where each non-leaf child contains another `Vector` object. To have all these consolidated into one data object would minimise overhead.
- Simplicity is a big factor, the developed tree structure over complicates the task. It is very easy to produce messy and fragile code when using it.

7.2 Identifier resolution

Once the symbol table is created, or in this case the symbol tree, unresolved identifiers can be resolved. For the resolution of a single AST node this can be described in pseudo code found in figure 7.2. In each of the following three figures null is returned if the identifier in question is unresolvable.

```
loop through definitions in the current block/scope
{ if(matching definition name has been found)
    return the definition
}
if(block is a method)
{ loop through parameter definitions
  { if(matching definition has been found)
      return the definition
  }
}
recursively search up the tree, stopping at the root node
return null
```

Figure 7.2: Pseudo code for identifier resolution

This was then extended to include resolution of methods, as is outlined in figure 7.3

```
loop through definitions in the current block/scope
{ if(matching method name has been found)
  { if(same number of parameters)
    { if(parameter types match)
        return the method
    }
  }
}
recursively search up the tree, stopping at the root node
return null
```

Figure 7.3: Pseudo code for method resolution

Furthermore there is resolution of multi-part names, which will be referred to a chain. The pseudo code for this resolution technique can be found in figure 7.4. This is probably the most complex operation as there are 4 possibilities of chain that could be passed to this method, these are:

1. A single AST wrapped as a vector. It handles results of the Tree Parser that have been handled unexpectedly by wrapping an AST node into a `Vector`. This should never happen but it is always best to err on the side of caution.

2. A chain of the form `this.variable`, where each element in the `Vector` is an AST node with the exception of the first element which may contain a String. The string could be Java commands such as `this`, or `super`, which need to be handled slightly differently. At the moment only `this` is supported.

3. A chain in the form `object.method()`, which should only contain two items. The first element is a chain of type 2 containing the name, and the second is a chain of the parameters being passed to the method in question.

4. The most complicated chain to disassemble, it is a `Vector` with 2 or more elements. All but the last element will be chains of type 3, and the last element could be type 3 or 2. In this case the method recursively calls itself to resolve each individual section of the reference, searching for a matching definition from the return type of the previous segment.

```
if(chain only has one element, and its an AST)
{ return the resolved AST node }
if(size > 1 and first element is a Vector)
{ //type = 4
  Scope s = this
  Definition part
  loop through the Vector
  { part = resolved segment starting at Scope s
    s = return type of the resolved segment
  }
}
else if(size==2 and both elements are Vectors)
{ type = 2 }
else
{ type = 1 }

if(first element = "this")
{ remove first element
  pass resolution up to nearest class Scope
}

loop through remaining chain
{ if(its a method)
  { resolve as a method
    return the definition found
  }
  else
  { resolve as an AST
    return associated definition
  }
}
return null
```

Figure 7.4: Pseudo code for multi-part identifier resolution

7.3 User Application

There were few problems of any interest encountered during the construction of the GUI as for the most part development proceeded smoothly. There is however the matter of loading a source file into the GUI. The first version of the application would freeze when loading the input file, making it look as if the program had crashed. This is undesirable in any application as it will confuse the user. This problem was solved by adding a second thread to the program which is given the file to load and then calls the necessary methods in the `JavaFile` class (in the API). Now that the GUI does not freeze, calls to start and stop the `JProgressBar` were added to the loading thread to inform the user that loading is in progress.

This solved the immediate problems and also will allow for a further extension to disable certain options within the GUI that would be undesirable while loading a file, opening another file for example.

One other small point to add is that when renaming an identifier with the current GUI, regular expressions are used to confirm that the variable name being entered is valid. Currently this is only a very loose implementation of of the naming conventions Java can employ, simply it only allows users to use alphanumeric names that cannot begin with a number.

Checking for correct identifier names is a feature that should have been moved into the refactoryAPI, preserving the centralised structure of the code logic. By implementing the check within the GUI it check becomes non-standardised and therefore opens the API to future bugs.

7.4 Reverse engineered UML diagrams

As there were no Unified Modeling Language (UML) diagrams produced during the system's design phase, diagrams of the current system have been reverse engineered by "Fujaba", a modelling application written in Java. Hence these diagrams belong under Implementation rather than Design. The diagrams in question can be found in the appendices on page 70. Each UML diagram presents an overview of the classes, and their relationships with other classes, found within their respective packages.

Chapter 8

Testing

This chapter will discuss the matter of testing the implemented system, firstly by comparing the features provided to the requirements outlined in the specification chapter. Secondly by running a set of test cases through the program, commenting on whether or not the output was expected. The test cases themselves and their output can be found in the appendix on page 80. Finally this chapter concludes with a summary and evaluation of these two types of testing.

8.1 Requirements Testing

This is a type of black box testing and involves looking back at the requirements and using them like a check list to confirm that everything that needed to be implemented within the system is present. Unless otherwise stated each requirement is considered to be covered correctly. Here are those requirements with explanations as to what was done to include each feature.

High priority:

1. Java version: Java1.4.2 was used
2. Relevant naming: As much as possible relevant names were used, as can be inspected by using the UML diagrams included in the appendix on page 70.
3. Protection: A majority of the methods are still public from the initial development phase, and for this reason the application has not achieved this requirement.
4. Source input: Only one file can be loaded, and parsed, at a time. However the user could attempt to load a second file, in which case a race condition is produced. The file which is parsed last is the file that is displayed within the GUI for manipulation.
5. Error handling: Reasonable error messages are logged to the user, and also outputted to a log file. The log file is less descriptive than it was intended to be, for example, when a variable has been renamed the log file simply indicates that the output file has been written. A few extra lines of code can be inserted to include more detailed logging of which identifier was renamed, and what its handle was changed to.
6. GUI: There is a GUI included that serves its purpose.
7. Display: Both the input and output files are displayed by use of the combo box selector at the bottom of the GUI. The current AST being operated on can be obtained through the menu ”view”.
8. Simple: All that is required is for the user to load a file and then right click on the definition they wish to rename. Once the identifier has been changed the GUI updates its display to the new output file.
9. Indication: A `JProgressBar` has been used to inform the user that a file is being parsed.
10. Source Safety: The output file is placed in a separate folder to the input file, and therefore does not overwrite the input file.

11. API: Unfortunately a lot of the logic that should be included within the API has been written into the GUI. This was not intentional, however as it is the application fails this requirement.

12. Refactoring: Renaming of both variables and method have been implemented to some extent, see the section below on test cases that have been run through the system.

13. Correctness: Aside from the issue of constructors, and data typing as discussed later, the output is correct in comparison the the input file.

8.2 Functional testing

Also a type of black box testing, functional testing examines an input to its output to ensure the program is working correctly. As can be seen in the figures found in the appendix on page 80, the system works under certain circumstances. More than the six scenarios listed have been tested, however it seems irrelevant to include them all. Only the six main situations have been included in the appendix which display the main behavioural traits of the system. The information that can be extrapolated from these test cases is contained in the following list.

Variables: This works as well as can be expected.

Methods: This works except when values are used as arguments that have not been wrapped in a variable. This means that when a method is called that contains data hard coded into the arguments, say an error message `String`, the method cannot be resolved. This issue can be dealt with by adding regular expression contextual rules to check what possible variable types could contain the data used, as there may be more than one. An example of this is the number `5.5`, it could either be stored as a `float` or a `double`, but not as an `int`.

Classes: Renaming of classes works correctly except for the matter of constructors. Constructors are not handled right now, but a relatively simple addition to method resolution could add a reference to the class to which the constructor belongs.

”this”: The ”this” operator correctly finds the definition required within its given scope, in other words it attempts to find a definition within the nearest class block.

8.3 Conclusion

As shown the system functions almost as expected, constructors for example were known to be an unimplemented feature. The use of values instead of declarations within

method arguments was an oversight however. It demonstrates that the program needs to be completely restructured and redesigned to make implementing such features more efficient. There is more of a discussion on this in the conclusions chapter, and more specifically the extensions section.

Chapter 9

Conclusions

This chapter concludes the project in two areas, project management and an evaluation of the system.

9.1 Project Management

The original work schedule and Work Breakdown Structure (WBS), as presented in the report outline, can be found in the appendix from page 89. This plan changed as it was soon discovered to be very ambitious, so non-essential tasks such as integrating automatic testing using the JUnit framework were dropped. Pressure of work, and understanding of the problem domain also slowed progress through the early stages. However during the Christmas holidays a lot of work was accomplished, time became available to focus only on the project. Specifically this time was spent learning ANTLR, and beginning the implementation of the system. This helped a great deal towards getting the project back onto schedule.

What follows is the list of objectives, as numbered within the Introduction, with descriptions as to how each objective had been met.

1. Research of refactoring techniques, this has been accomplished by use of the references found in the bibliography. Compiler design has also been researched as a refactoring program implements translation from and to the same language. Terence Parr agrees with this as can be seen in his email correspondence in the appendix on page 88. This objective has been completed.

2. Both ANTLR and LaTeX have been used for the first time within this project, and have been learned to a high degree. Therefore this objective has been accomplished.

3. Functional and non-functional requirements have been defined, see the Specification chapter for further details, therefore this objective has been met.

4. The system was not designed as adequately as it should have been, this is because there was an overwhelming amount of research that had to be conducted and summarised. Therefore this objective was not reached.

5. The system was implemented, the problems encountered are outlined in the Implementation chapter, with solutions where appropriate. Although not to the standard as first intended, this objective has been completed.

6. Complete tests, such as unit testing, were not performed as implementing the system finished so close to the deadline. However some testing has been summarised in this report, such as functional testing for example. The nature of refactoring implies constant testing, and much of this was done using test cases produced on the fly. Although not as thorough as hoped it can be concluded that this objective has been met.

7. A final report has been written and submitted by the deadline as stated.

Project management was one of the hardest tasks during this project but was essential to ensure completion on target. With hindsight it could have been improved with better planning and more effective initial research. Time management in particular is a challenge when there are many conflicting tasks to prioritise. Learning such skills is often best achieved by practice and this project provided the opportunity for developing and advancing expertise in project management

9.2 Evaluation and Extensions

This project has some areas of functionality that need improving and extending to remedy the majority of the problems found. However in the end the system design is flawed. Many of the issues raised during testing and general examination of the system whilst writing this report can be attributed to the symbol tree. It over-complicated a lot of the code and made it much more difficult to extend in its current form than it could have been. An example of this is the use of nested `Vectors` being passed through the system, which are inefficient when used in such a manner.

The following is a list of changes to the program, including possible extensions.

- Change the system to use a symbol table rather than a symbol tree as it has been discovered to be less efficient and generally more trouble than the alternative. At the very least the ability to flatten the symbol tree into a table before any resolution is performed should be written. Also an interface should be used to ensure that differently implemented symbol table objects will contain the same methods and behave in a predefined way.

- Remove the Javac check, as it reduces compatibility and performance of the system. Replace it with integrated semantic checks generated from an easily editable source. For example these semantic rules could be written in XML, or other context language, that can be loaded into the program. This with a strongly defined symbol table interface would provide much greater flexibility, maybe into other languages. It could be possible to make a ”parser pack” where, for example, grammar files and XML formatted semantic rules could combined together in a zip file package. These packages could then be extracted and installed automatically into the program's directory structure and therefore integrated into the system.

- An object oriented database could be used, such as DB40[Inc04], to store ASTs of unchanging files, such as files integral to Java, so they do not have to be reparsed each time the program loads. In the case of DB40 these databases can be dumped to a file, which could then be added to the distribution packages as outlined in the above comment.

- Advance the program to cover all features of Java1.3, and then extend that further to use Java1.5. This again can fit with the package idea, as different versions of Java could be contained in separate packages and the user could use the one which best suits them at will.
- A scramble feature could be produced, to randomly generate identifier names. This would prevent the Reflection API from being able to extract understandable information, unless the symbol table were obtained and used as a decryption key.
- The system could also be used to diagrammatically display the input source code, rather than simply displaying plain text, in a format analogous to UML. At the very least syntax highlighting could be used when displaying the code, perhaps even with automatic selection of all references to an item in the symbol table.
- Include comments in the outputted code, and while retaining the original formatting is near impossible, allow the user to be able to select how they wish their code to be exported from the AST. This advances the program to have code beautifier features.
- The GUI needs to be reimplemented to account for the changes to the system as proposed above, but also to add in the functionality currently missing. For example disabling options whilst parsing a file, whose use would be unwise during that time.

9.3 Final summary

This technical project has covered the issues involved with implementing refactoring techniques on Java source code. The system designed and built as described over the past few chapters used a symbol tree, in comparison to the more traditional symbol table, to store identifier definition information. This has been concluded to be the least efficient of the two options, and all future improvements involve updating the system to operate with the table data structure.

Learning has been considerable during the time spent on this project, and despite the problems along the way the overall goal, to implement refactoring techniques in the Java language, has been achieved.

Bibliography

[D.M01] R.Wilhelm & D.Maurer. *Compiler Design*.
Addison-Wesley Publishing Company,
print on demand edition, 2001.
A very detailed book concerning compiler design and construction. It covers programming languages other than Object Oriented software, such as Functional and Logic languages. Although that is good to get an general grasp of compiler design it does not provide as much detail as I would like for the context of this project. There is however a lot of very useful pseudo code examples used to explain different sections of the book and make it easier to grasp.

[G.G05] G.Garcia. http://wiki.java.net/bin/view/people/smellstorefactorings,
March 2005.
A good list of refactoring 'bad smells' from the book [FOW1], it includes explanations of some of the different refactoring transformations.

[Inc04] DB4Objects Inc. http://db4o.com,
November 2004.

[I.S01] I.Sommerville. *Software Engineering*.
Addison-Wesley Publishing Company,
6th edition, 2001.
A good software engineering book used for information during the early stages of development.

[J.B05] J.Byous. http://java.sun.com/features/1998/05/birthday.html,
March 2005.
One of the many good pages found on the Sun Microsystems website, provides quite an in-depth history of Java and links to alternate versions found on such sites as Wired.

[J.G00] J.Gosling. *Java Language Specification, The*.
Addison-Wesley Publishing Company,

2nd edition, 2000.
An online version of this book can be found on the Sun Microsystems website, a copy of the first edition is also available here. The third edition is also likely to appear here once it is released. This book is invaluable when writing programs that work so closely with the Java syntax as a refactoring program does. It has helped to clear up many questions about fine details in rules, especially in the case of scope.

[J.S63] R.A.Brooker & I.R.MacCallum & D.Morris & J.S.Rohl.
the compiler compiler.
Annual review in automatic programming, 3:53, 1963.

[K.K91] E.Rich & K.Knight. *Artificial Intelligence*.
McGraw-Hill, Inc.,
international edition, 1991.
This book is about Artificial Intelligence and therefore almost completely off-topic. However there are a few sections of interest, namely Pages 387 and 388 where a short and concise argument of the benefits and limitations of Top-Down parsing against those of Bottom-Up.

[M.A05] M.Atkinson. http://jrefactory.sourceforge.net,
March 2005.
This is the homepage to the JRefactory program. Personally I don't like this page, although it provides a lot of information about the program, there is not one specific statement to say that 'JRefactory is a suite of tools', which is what I assume it is. This would not be so bad if one of the tools within JRefactory were not called JRefactory.

[M.F04] M.Fowler. *Refactoring: Improving the Design of Existing Code*.
Addison-Wesley Publishing Company,
14th edition, 2004.
This book is very interesting, and quite easy to read. It is mainly based in the theory of refactoring and makes use of lots of examples. These examples are clear and are to the theme of the book. It is written by and has contributions from some of the leading people in this field.

[M.F05] M.Fowler. http://www.refactoring.com,
March 2005.
A page maintained by Martin Fowler with all sorts of useful information and tools to help refactor your code or understand refactoring it better.

[oK05] University of Kent.
http://www.cs.kent.ac.uk/research/pg/tcs.html,

March 2005.
Examples of active research projects in the field of Refactoring.

[P.W04] P.William. http://www.smalltalk.org/smalltalk/history.html,
November 2004.
This site provides a lot of information regarding the SmallTalk language, including history, online books, links and downloadable versions for your computer.

[R.Q96] T.Parr & R.Quong.
ll and lr translator need k.
SIGPLAN Notices, 31#2, 1996.

[S.V04] S.Viswanadha. https://javacc.dev.java.net,
November 2004.
This is the homepage of the JavaCC (Java Compiler Compiler) Parser Generator. Along with one of its plugin programs JTree these programs can parse Java source code into things like AWT trees.

[T.P05a] T.Parr. http://www.antlr.org,
March 2005.
This it the homepage of the parser generator that I used to construct my application, specifically version 2.7.5.

[T.P05b] T.Parr.
http://www.antlr.org/doc/glossary.html,
March 2005.
A very useful page with glossary items relevant to ANTLR software.

[T.P05c] T.Parr.
http://www.jguru.com/faq/view.jsp?eid=83,
March 2005.
about 4 useful words on this particular page, however the JGuru site has been indispensable during the course of this project and report. It is a great resource to new and experienced Java programmers alike.

[T.S99] T.Parr & G.Voss & J.Coker & S.Stanchfield & T.Sundsted.
http://java.sun.com/developer/technicalarticles/parser/index.html,
1997 - 1999.
(Accessed March 2005) This is a set of very useful articles introducing ANTLR and talking about Symbol Table generation. It is old now as most of the code is written for Java1.3. However the theory is still sound. The only problem is that a few sections seem to jump and the assumptions of

knowledge can occasionally change dramatically. Maybe it is because the majority of the articles are written by more than one author and occasionally the writing styles do not meet evenly.

[Unk05a] Unknown. http://st-www.cs.uiuc.edu/users/brant/refactory, March 2005.
Web page with the features of the 'Refactoring Browser' application includes a link to the most up to date information on the Refactoring Browser.

[Unk05b] Unknown.
http://transmogrify.sourceforge.net/,
March 2005.
See the Developers Guide for technical information. Used for reference in that is an application with goals similar to the aim of this project and uses ANTLR to accomplish it.

Glossary

Abstract Syntax Tree (AST)

A representation of output from a Parser showing syntax in a tree structure.

ANTLR

ANother Tool for Language Recognition - written by Terence Parr, please see the ANTLR website for more information.

Application Program Interface (API)

A published interface to a program, once created it should be treated as a contract between the users and programmers. If anything needs to be altered it should first be Depreciated to warn users of its change.

Backus Naur Form (BNF)

Formal notation for defining the syntactic structure of a language.

Bad smells

A trait found in source code that could lead to bugs later in program development.

Compiler Compiler (CC)

Also known as Parser Generators, a Compiler Compiler is a program designed to take a grammar file as input and produce a parser/recognizer.

Graphical User Interface (GUI)

A graphical method of interacting with a program. In Java this would use either the AWT or Swing packages.

Inheritance

A feature of Object Oriented languages, where an Object can be extended to add greater more defined functionality, such as extending a person class into an employee class.

Integrated Development Environment (IDE)

A literal definition is any application that provides access to a set of tools such as a compiler, code editor and GUI builder, for the purposes of aiding the creation of software. Also known as an *Interactive Development Environment*. Common examples are IntelliJ, Eclipse, and Microsoft Visual Studio(.net)

Java Development Kit (JDK)

Includes all features of the JRE. Also includes a suite of basic programs used for Java software development that can be used on their own or integrated into other programs, such as an IDE. These tools include: javac (compiler), and javadoc (documentation tool).

Java Server Pages (JSP)

A type of server side scripting using Java for creating powerful dynamic content.

Java Virtual Machine (JVM)

The main piece of software required to execute a Java program, executing Java bytecode.

Lexical Analysis

Reads in a stream of characters, and creates sets of more readable Lexical Units.

Lexical Unit

Often known as a symbol, a token, or a word.

Object Oriented (OO)

A type of programming language, similar to Functional languages, where the programming logic is separated into classes (objects), an example of which is Java. Objects consist of a set of variables and methods, for example a Person object might have a name attribute with getter and setter methods. OO programming languages also allow for inheritance.

Operating System (OS)

The low-level software which handles peripheral hardware, schedules tasks, allocates memory, and presents a default interface to the user.

Parser Generator

See *Compiler Compiler*

Portable Document Format (PDF)

A file type commonly associated with Adobe Acrobat, it comes from the Postscript file family.

Refactoring

A transformation on a piece of source code that when used in sequence can make drastic changes to the internal structure of a program, without changing its external interface.

Semantic Analysis

Ensures that the result of Syntactic Analysis complies to context conditions. Examples of context conditions are Scoping and Typing rules.

Sentence

see *Syntactic Unit*

Standard Development Kit (SDK)

See JDK

Syntactic Analysis

Reads in a stream of tokens, and attempts to match them to a predefined set of rules usually provided by a grammar, creating Syntactic Units.

Syntactic Units

A set of tokens that have matched a rule found in a grammar, often known as a sentence.

Transformations

See *Refactoring*

Unified Modeling Language (UML)

A standardised specification for modelling an applications architecture and behaviour. On a side note it can also be used to model processes in businesses.

Word

see *Lexical Unit*

Work Breakdown Structure (WBS)

A method of breaking down the objectives of a task into smaller tasks, this information can then be converted into a work schedule.

Appendices

The following pages contain information and figures that are supplementary to the report. Every effort has been made to make the text in the images presentable and readable, while conserving space.

Appendix A - CD contents

Here is a complete listing of the files found on the CD:

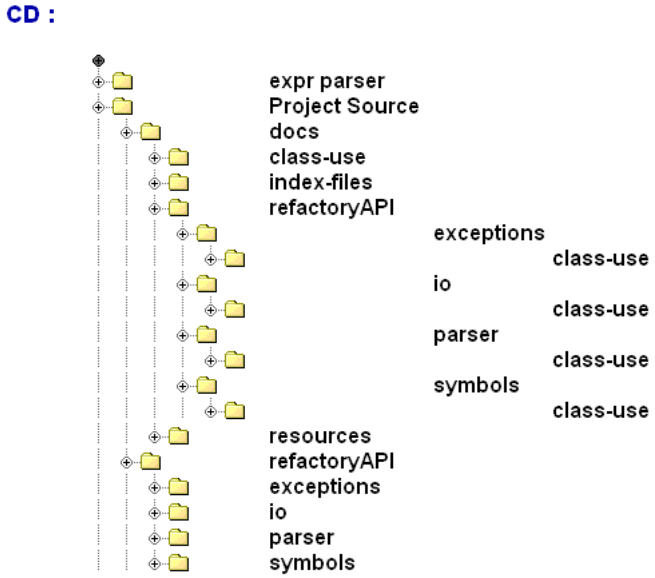


Figure 9.1: Appendix A - CD directory hierarchy

The root of the CD contains both the report and abstract in PDF form, along with a zip file containing the final report's source code in LaTeX format, and another zip file which contains a complete copy of the Eclipse workspace used during development in case the program needs to be executed. In the directory "Project Source" the source code and Javadoc files of the main application can be found. The directory "expr parser" contains the source code to the expression parser as outlined in the chapter Learning ANTLR. The file `Main.java` is based on the file of the same name as found with the grammar files listed below in the ANTLR distribution, modified for use in this context. All the grammar files within this directory are my own work. Also included are two windows batch scripts, `c.bat` used to compile the grammar files, and `r.bat` used to run the program using the file `input` as an argument.

Also included in this appendix is a set of files which were authored by people other than myself, Jonathan Nicholson. If they were modified in any way details of this will be given. All these files are contained in the current ANTLR release (version 2.7.5) in the public domain. For information pertaining to the authors of these files, please see the list contained in `Java.g`.

- `Project Source/refactoryAPI/parser/java.g`
  This has been modified only so that the initial comment is Javadoc compatible. The following files are generated from this grammar:

`JavaLexer.java`,
`JavaRecogniser.java`,
`javaTokenTypes.java`, and
`JavaTokenTypes.txt`

- `Project Source/refactoryAPI/parser/java.def.g`
  Any Java code found herein is my own work, but the rules themselves are not.
  The following files are generated from this grammar:
  `JavaDefMaker.java`,
  `JavaDefMakerTokenTypes.java`, and
  `JavaDefMakerTokenTypes.txt`

- `Project Source/refactoryAPI/io/JavaEmitter.g`
  This has been modified only so that the initial comment is Javadoc compatible.
  The following files are generated from this grammar:
  `JavaEmitter.java`,
  `JavaEmitterTokenTypes.java`, and
  `JavaEmitterTokenTypes.txt`

Appendix B - Expression parser code

The main grammar file containing the Lexer and Recognizer, it produces MathLexer and MathParser.

```
header {
/*
 * Written by Jonathan Nicholson
 *
 * So far operator precidence is to the left, and ignores differences
 * if its * or /. Brackets change this flow and enforce precidence.
 *
 * doesnot know how to deal with things in the form 2x
 */

}

class MathParser extends Parser;

options {
    k = 4;
    exportVocab=Math;
    buildAST=true;
}

{
    // Hashtable for the symbol table (symtab)
    private java.util.Hashtable symtab = new java.util.Hashtable();

    public java.util.Hashtable getSymTab()
    {
        return symtab;
    }
}

// Initial rule used to parse the file
compile
    : ((decl)+)? begin EOF!
    ;

// Declarations, don't add these to the tree
decl!
    : l:LITERAL ASSIGN n:NUMBER SEMI
    {
       symtab.put(
          l.getText(),
          new Double(Double.parseDouble(n.getText()))
          );
    }
    ;
```

```
// Rule matching the equation to solve
begin
   : ASSIGN^ expr
   ;

// Deals with infintaly long expressions in the form:
// X1 + X2 + ... + Xn-1 + Xn, and numbers on their own
expr
   : atom ((OP^ atom)+)?
   ;

// Atomic sections of the tree
atom
   // Negative expressions
   : OP^ atom

   // An expression in brackets
   | LPAREN expr RPAREN

   // Literal character variables
   | LITERAL

   // Numbers
   | NUMBER
   ;

/** * * * * * * * * * * * * * * * * * * * * * **/
/** * * * * * * * *Lexer* * * * * * * * * * **/
/** * * * * * * * * * * * * * * * * * * * * * **/

class MathLexer extends Lexer;

// Whitespace token, ignore them
WS
   : (' ' | '\t' | '\n' | '\r')
   {
     _ttype = Token.SKIP;
   }
   ;

// Semi Colon for declarations
SEMI
   : ';'
   ;

// Bracket tokens
LPAREN
   : '('
   ;
```

```
RPAREN
   : ')'
   ;

// Assignment operator
ASSIGN
   : '='
   ;

// Operators
OP
   : PLUS
   | MINUS
   | DIV
   | MUL
   ;

// Literal characters for variables
LITERAL
   : 'a'..'z'
   | 'A'..'Z'
   ;

// Numbers (handles ints, floats, doubles, etc)
NUMBER
   : (DIGIT)+ (DEC (DIGIT)+)?
   ;

/**
 * Basic types that are extended on above
 **/

protected
DEC
   : '.'
   ;

protected
PLUS
   : '+'
   ;

protected
MINUS
   : '-'
   ;

protected
DIV
```

```
    : '/'
    ;

protected
MUL
    : '*'
    ;

protected
DIGIT
    : '0'..'9'
    ;
```

TreeParser to solve the given expression and gets values for any variables not yet defined, produces MathTreeParser

```
header {
/*
 * Written by Jonathan Nicholson
 */

import java.io.*;
}

class MathTreeParser extends TreeParser;

options {
    k = 4;
    importVocab = Math;
}

{
    // Hashtable for the symbol table (symtab)
    private java.util.Hashtable symtab = new java.util.Hashtable();

    public void setSymTab(java.util.Hashtable st)
    {
        symtab = st;
    }

    public java.util.Hashtable getSymTab()
    {
        return symtab;
    }

    // Method to resolve the operator chars into binary actions
    private double binary(String x, double a, double b)
    {
        if(x.equals("+"))
            return a+b;
        else if(x.equals("-"))
            return a-b;
        else if(x.equals("*"))
            return a*b;
        else if(x.equals("/"))
        {
            // Return 0 if its divide by 0
            if(b==0)
                return 0;
            else
                return a/b;
        }
        else
```

```
            return 0;
        }

    // Method to resolve the operator chars into unary actions
    private double unary(String x, double a)
    {
        if(x.equals("+"))
            return +a;
        else if(x.equals("-"))
            return -a;
        else
            return 0;
    }
}

// Rule returns a double value
begin returns [double value = 0]
    : #(ASSIGN value=expr)
    ;

expr returns [double value = 0]
    {
        // Make temp variables
        double left = 0;
        double right = 0;
    }

    // The main recursive rule to match the tree
    : #(a:OP left=expr (right=e:expr)?)
    {
        // Deals with unary or binary operators
        // allows infinate unary operators, eg --+-+1
        if(e==null)
            value = unary(a.getText(), left);
        else
            value = binary(a.getText(), left, right);
    }

    // Could be an atomic section of tree...
    | value=atom
    ;


// Atomic sections of the tree
atom returns [double value = 0]
    // Something surrounded by brackets
    : LPAREN value=expr RPAREN

    | l:LITERAL
    {
```

```
        String s = l.getText();

100     if(symtab.get(s) != null)
        {
            Double val = (Double)symtab.get(s);
            value = val.doubleValue();
        }
105     else
        {
            // Get value from user
            String question = "Please enter a value for " + s + ": ";

110         // Print the question
            System.out.print(question);

            // Make an input reader
            BufferedReader in = new BufferedReader(
115             new InputStreamReader(System.in)
                );

            String line = "";
            Double D;
120
            // Get a value for the variable
            readInput:
            while(true)
            {
125             try
                {
                    line = in.readLine();
                    D = new Double(line);

130                 // Put the symbol into the SymTab
                    symtab.put(s, D);

                    // return it
                    value = D.doubleValue();
135
                    // all is good so break the loop
                    break readInput;
                }
                catch(Exception e)
140             {
                    /*
                     * If they dont want enter a value then
                     * exit the loop and store zero
                     */
145                 if(line.equals("exit"))
                    {
                        symtab.put(s, new Double("0"));
```

```
                    break readInput;
                }
                else
                    System.out.print(question);
            }
        }
    }
}

// An integer
| i:NUMBER
{
    value = Float.parseFloat(i.getText());
}
;
```

This is a TreeParser to replace the variable with its value for every point it appears in the tree, it must have a complete symbol table to do this however. Which means that MathTreeParser should be executed first. This feature was separated from the MathTreeParser class so that the logic is clearly visible. It produces MathTreeChanger.

```
header {
/*
 * Written by Jonathan Nicholson
 */

import java.io.*;

}

class MathTreeChanger extends TreeParser;

options {
    k = 4;
    importVocab = Math;
}

{
    // Hashtable for the symbol table (symtab)
    private java.util.Hashtable symtab = new java.util.Hashtable();

    public void setSymTab(java.util.Hashtable st)
    {
        symtab = st;
    }

    public java.util.Hashtable getSymTab()
    {
        return symtab;
    }
}

// Rule to deal with root nodeand its sub tree
begin
    : #(ASSIGN expr)
    ;

expr
    // The main recursive rule to match the tree
    : #(OP expr (expr)?)

    // Could be an atomic section of tree...
    | atom
    ;
```

```
45
// Atomic sections of the tree
atom
    // Something surrounded by brackets
    : LPAREN expr RPAREN
50
    // A variable
    | l:LITERAL
    {
        String s = l.getText();
55
        if(symtab.get(s) != null)
        {
            Double value = (Double)symtab.get(s);
            l.setText(value.toString());
60          l.setType(NUMBER);
        }
        else
        {
            System.out.println(
65              "Sorry, the symbol table seems to be incomplete"
                );
        }
    }

70  // An integer
    | NUMBER
    ;
```

This is the main application class that brings the ANTLR generated code together. It displays two windows detailing the contents of the AST, one after executing MathLexer, MathParser and MathTreeParser. The other after executing MathTreeChanger. The difference between the two windows should be that the first includes LITERAL (variable) nodes, which will have been replaces with the number they represent in the latter.

```
/*
 * Based on the Main.java file found with the Java1.3 grammar
 */

import java.io.*;
import antlr.collections.AST;
import antlr.debug.misc.*;
import antlr.*;
import java.awt.event.*;

class Main
{
    static boolean showTree = false;
     public static void main(String[] args)
     {
        try {
            // loop through arguments, parsing each file given
            if (args.length > 0 ) {
                System.err.println("Parsing...");
                for(int i=0; i< args.length;i++) {
                    if ( args[i].equals("-showtree") ) {
                        showTree = true;
                    }
                    else {
                        doFile(new File(args[i]));
                    }
                } }
            else
                System.err.println("Usage: java Main [-showtree] "+
                                   "<directory or file name>");
        }
        catch(Exception e) {
            System.err.println("exception: "+e);
            e.printStackTrace(System.err);
        }
    }


    // Parses the file
    public static void doFile(File f) throws Exception
    {
        parseFile(f.getName(), new BufferedReader(new FileReader(f)));
```

```
    }

45  // Here's where we do the real work...
    public static void parseFile(String f, Reader r)
                            throws Exception
    {
        try {
50          // Create a Lexer
            MathLexer lexer = new MathLexer(r);
            lexer.setFilename(f);

            // Create a Recognizer
55          MathParser parser = new MathParser(lexer);
            parser.setFilename(f);

            // start parsing at the compilationUnit rule
            parser.compile();
60
            // do something with the tree
            doTreeAction(f, parser.getAST(), parser.getTokenNames(), parser);
        }
        catch (Exception e)
65      {
            System.err.println("parser exception: "+e);
            e.printStackTrace();
        }
    }
70
    public static void doTreeAction(String f, AST t, String[] tokenNames, MathParser parser)
    {
        if ( t==null ) return;

75      if ( showTree )
        {
            double result = 0;

            // Make new instances of the TreeParsers
80          MathTreeParser tparse = new MathTreeParser();
            MathTreeChanger cparse = new MathTreeChanger();

            // Make an new ASTFactory instance
            ASTFactory factory = new ASTFactory();
85
            try
            {
                // Copy the symbol table
                java.util.Hashtable symTab;
90              symTab = parser.getSymTab();
                tparse.setSymTab(symTab);
```

```
        // Get the result
        result = tparse.begin(t);

        // Print the now completed symbol table
        System.out.println("----Printing the Symbol Table----");
        System.out.println(tparse.getSymTab().toString());
        System.out.println("----          Completed      ----");

        // Get the completed symbol table
        symTab = tparse.getSymTab();
        cparse.setSymTab(symTab);

        // COPY the tree and change the COPY
        AST _t = factory.dupTree(t);

        // Modify the AST
        cparse.begin(_t);

        // Display the ASTs
        // NOTE: Closing one window will close both

        // Output the original AST
        ((CommonAST)t).setVerboseStringConversion(true, tokenNames);
        AST r = factory.create(0,"result = " + result);
        r.setFirstChild(t);
        final ASTFrame frame = new ASTFrame("Math AST", r);
        frame.setVisible(true);
        frame.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing (WindowEvent e)
                {
                    frame.setVisible(false);
                    frame.dispose();
                    System.exit(0);
                }
            }
        );

        // Output the modified AST in another window
        ((CommonAST)_t).setVerboseStringConversion(true, tokenNames);
        AST cr = factory.create(0,"result = " + result);
        cr.setFirstChild(_t);
        final ASTFrame cframe = new ASTFrame("Modified Math AST", cr);
        cframe.setVisible(true);
        cframe.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing (WindowEvent e)
                {
```

```
                        cframe.setVisible(false); // hide the Frame
                        cframe.dispose();
                        System.exit(0);
                    }
                }
            );
        }
        catch (RecognitionException e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
        }
    }
}
}
```

Appendix C - Reverse engineered UML diagrams

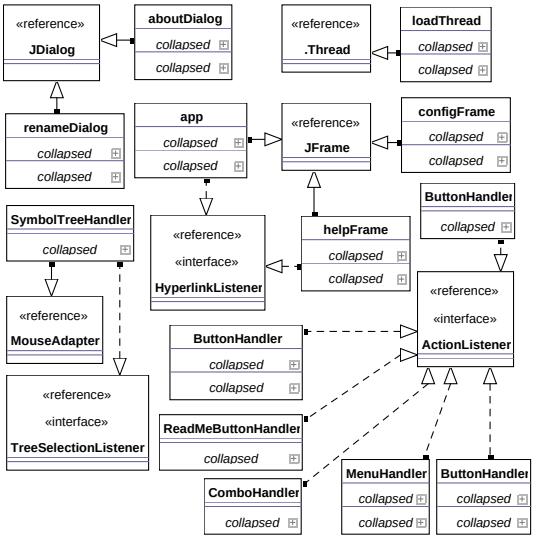


Figure 9.2: Appendix C - UML diagram for the GUI

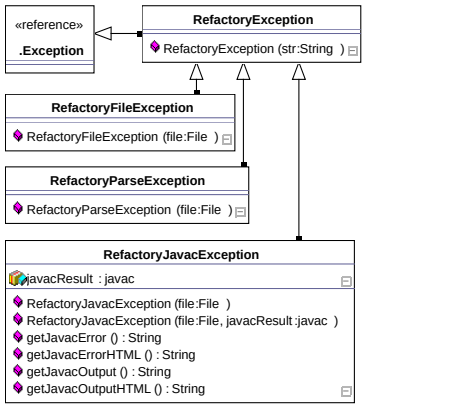


Figure 9.3: Appendix C - UML diagram for the exceptions package

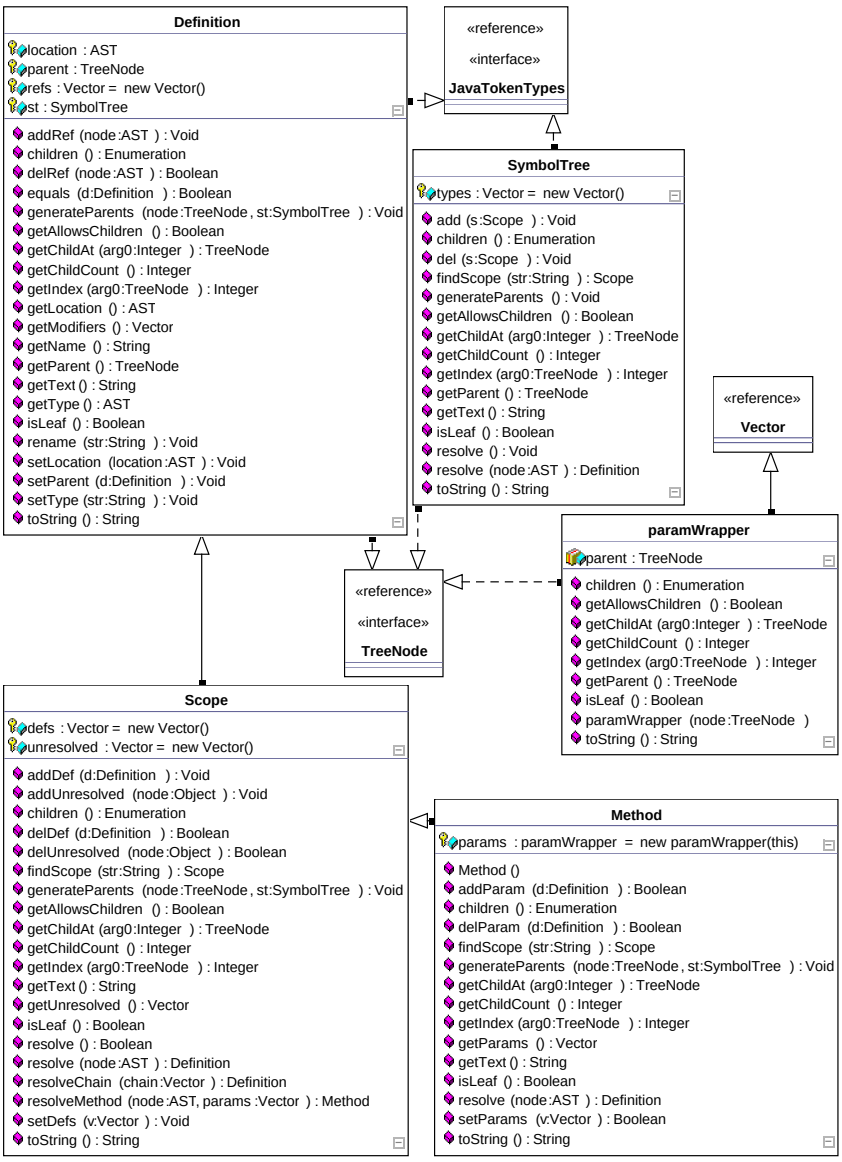


Figure 9.4: Appendix C - UML diagram for the symbols package

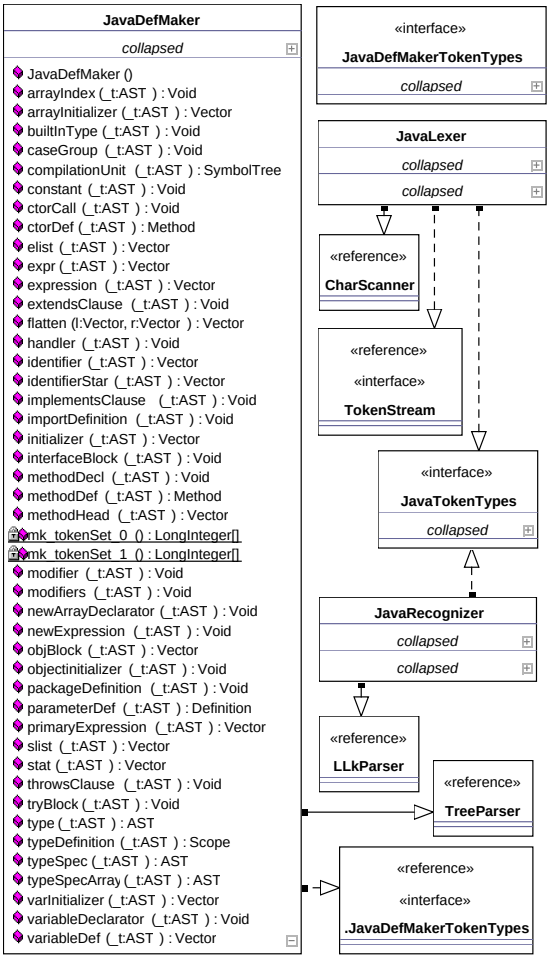


Figure 9.5: Appendix C - UML diagram for the parser package

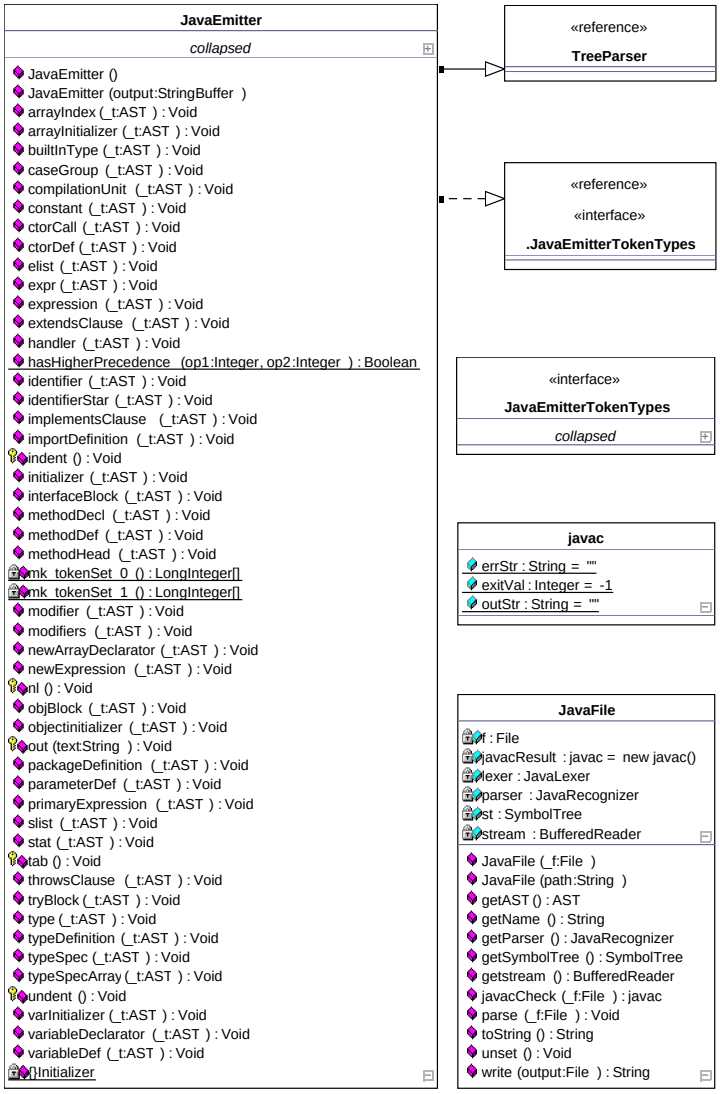


Figure 9.6: Appendix C - UML diagram for the io package

Appendix D - GUI design

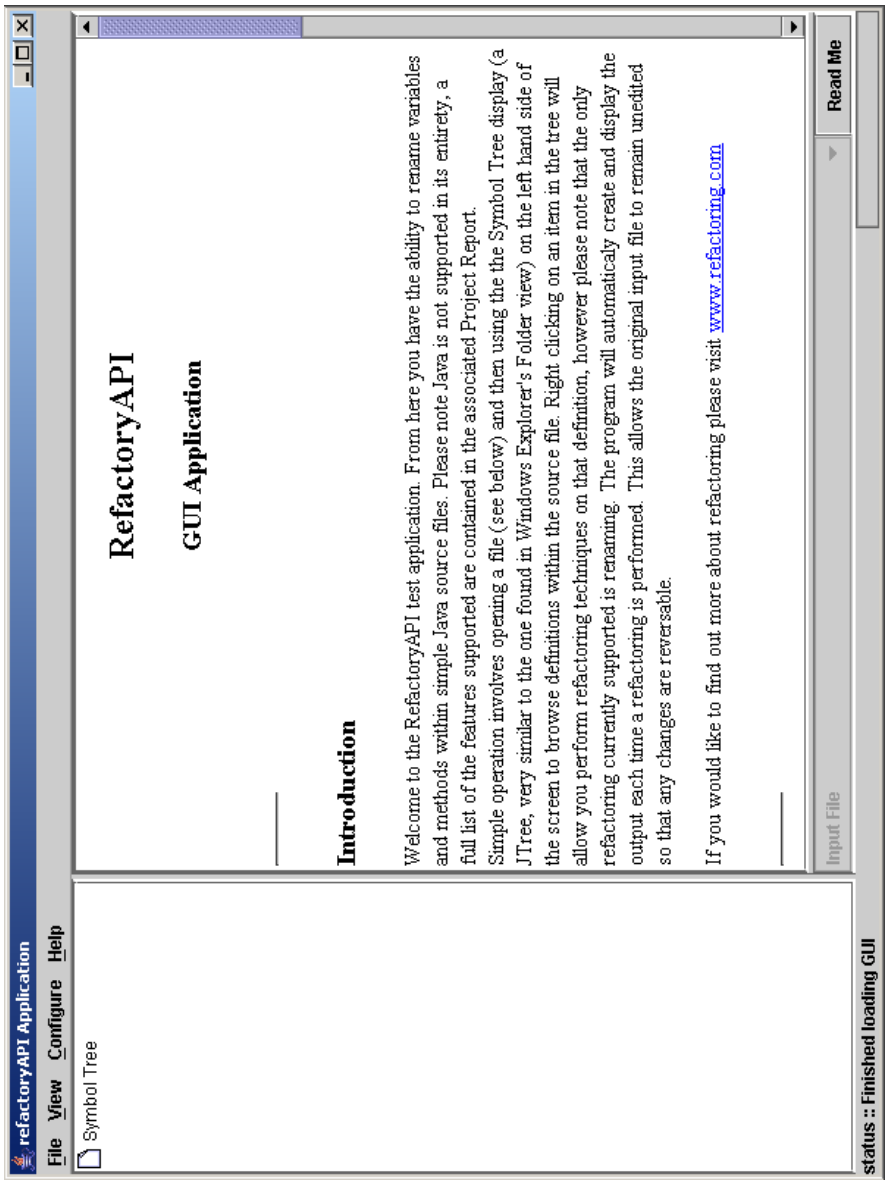


Figure 9.7: Appendix D - GUI's initial screen

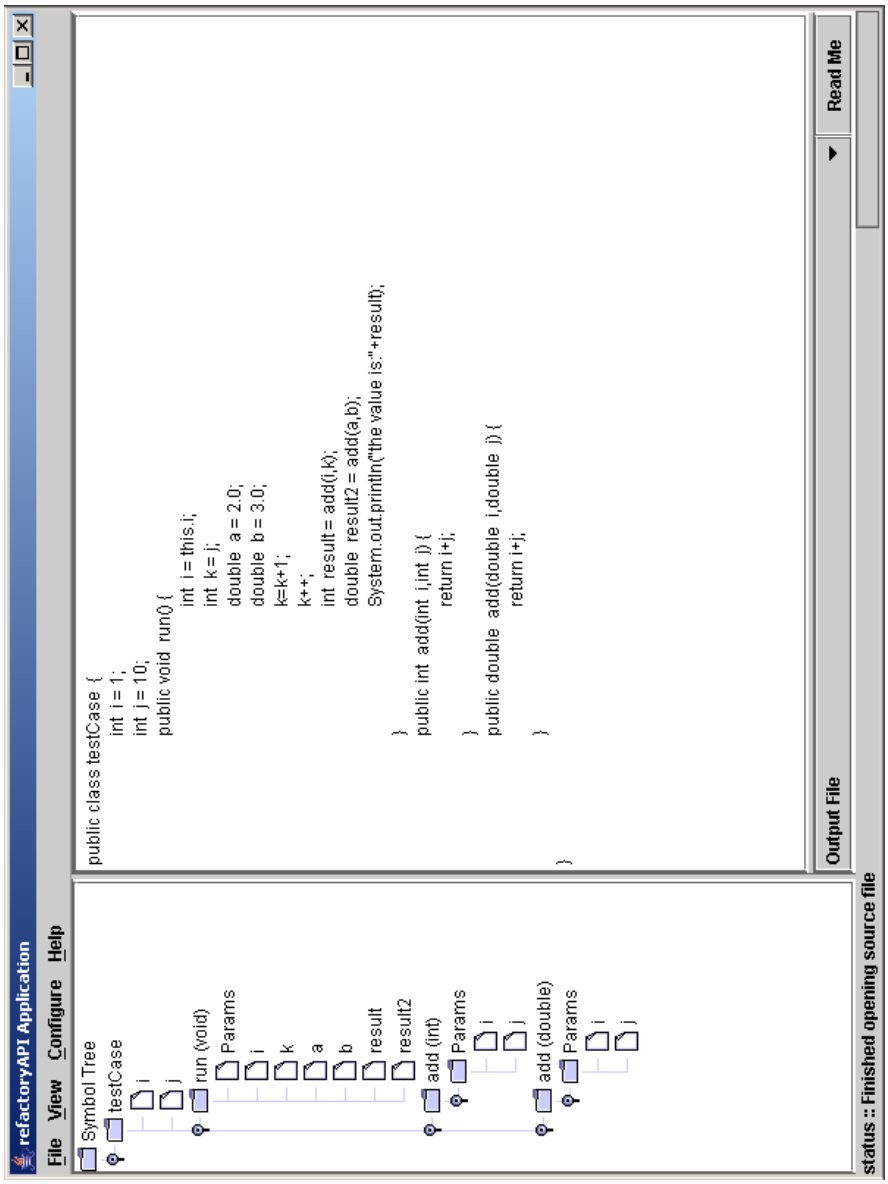


Figure 9.8: Appendix D - GUI after loading a Java source file

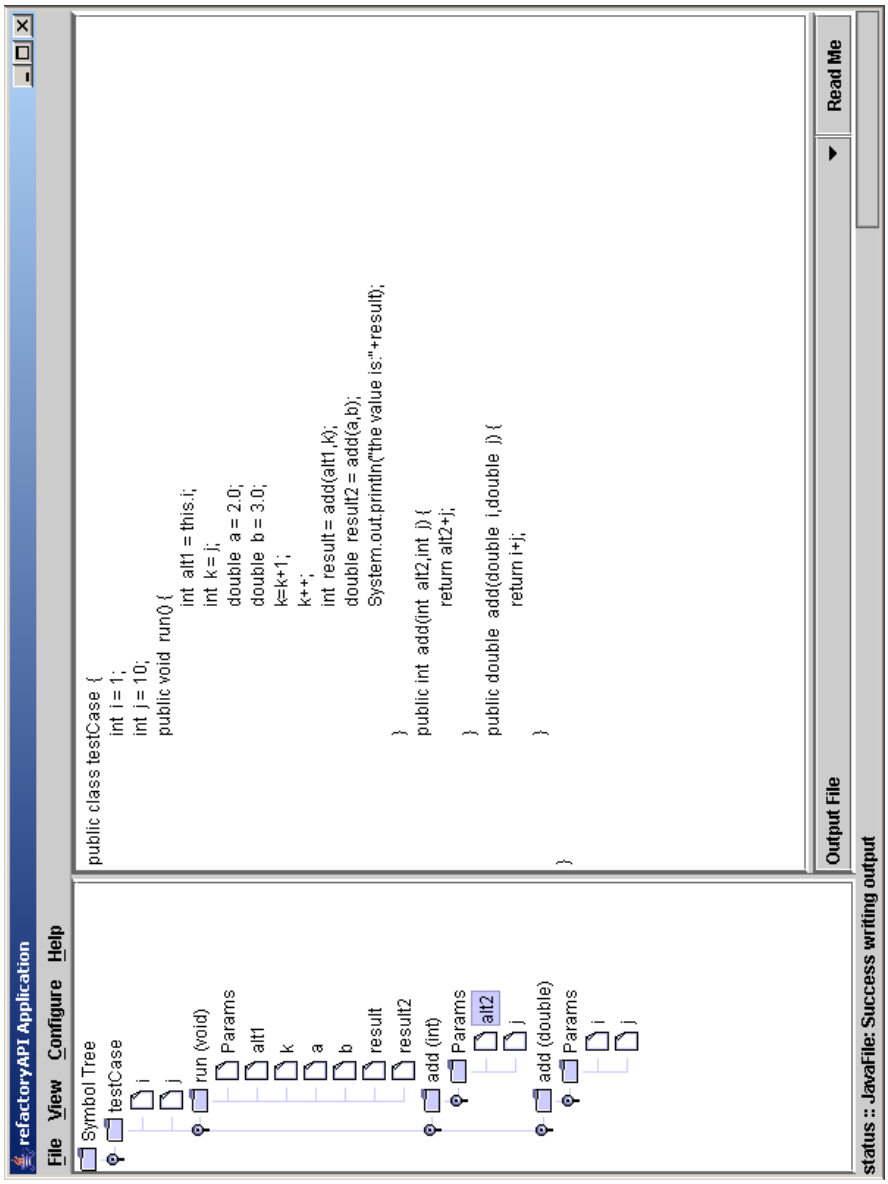


Figure 9.9: Appendix D - GUI after renaming refactoring

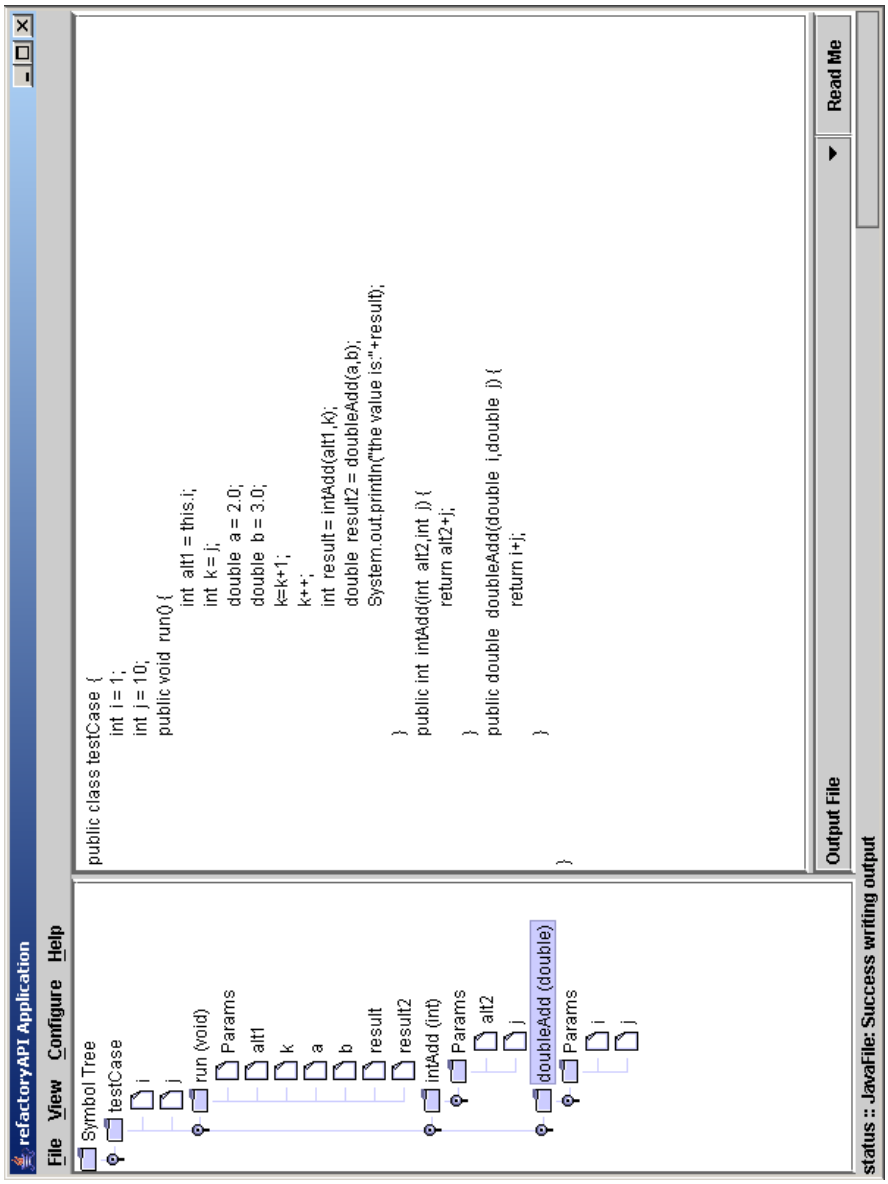


Figure 9.10: Appendix D - GUI after method renaming

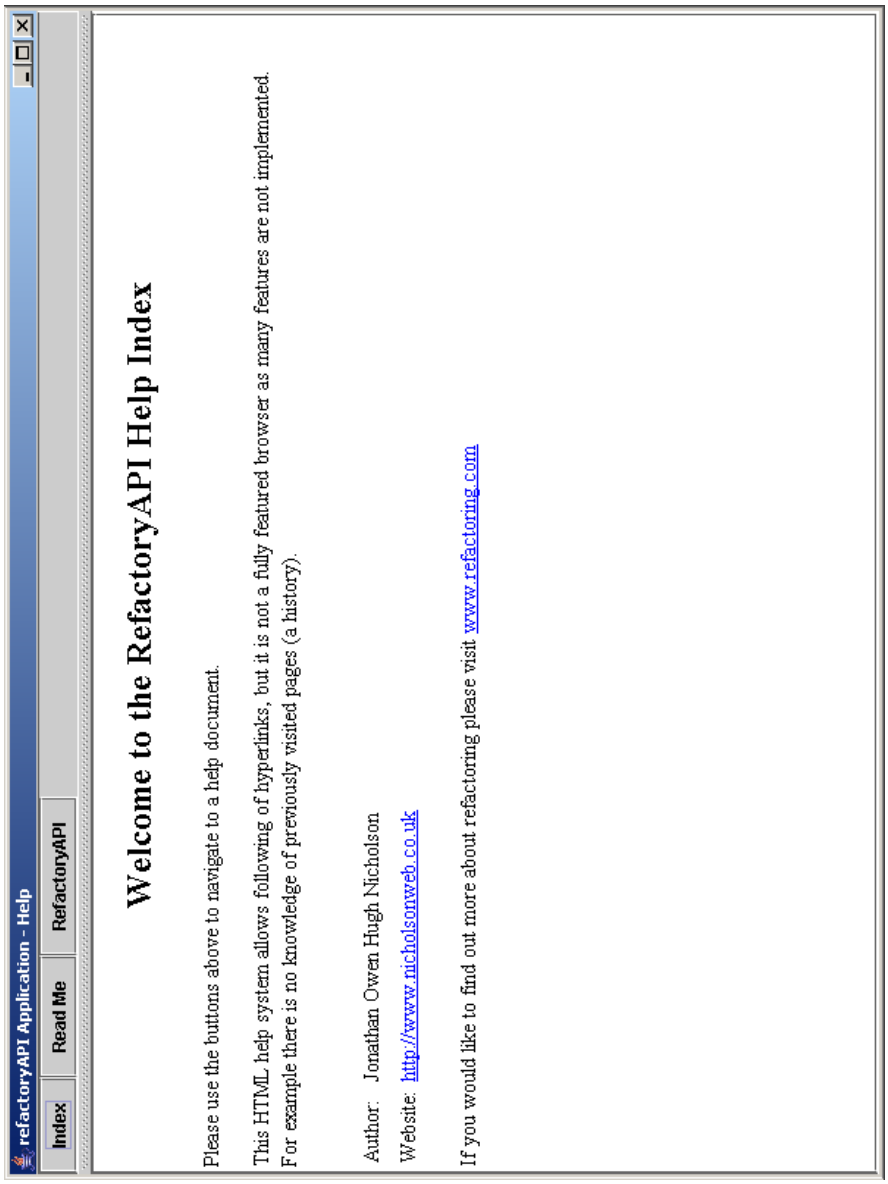


Figure 9.11: Appendix D - GUI help screen

Appendix E - Complete list of applications used

What follows is a fairly complete list of the applications used during this project with detailed information as necessary.

ANTLR: Parser Generator, version 2.7.5

Java: The programming language used, version 1.4.2. Includes the use of the JavaDoc application.

Eclipse: The IDE used to develop all the software in this project. Perhaps its biggest advantage is that it is free. Version 3.0.1. Also used with this application is the Jigloo plugin, used to create the GUI, version

MiKTeX: Suite of LaTeX tools, version 2.4.1779

TeXnic Center: The LaTeX IDE used to write this report, version 1 beta 6.21 ('fawkes')

Acrobat Reader: Program used to view the output of the files written using the TeXnic Center. Version 7.0.0

Fujaba: Application used to reverse engineer UML diagrams from source code, version 4.2.0 RE, build 20041122

HTMLDirCreate: Application used to generate the directory structure image found in Appendix A, page 54, version 2.4.4

Appendix F - Test Cases

Testing without any references:

```
public class testcase1
{
    int i;
    int j;

    public int add()
    {
        return 0;
    }
}
```

Figure 9.12: Appendix F - Test Case 1 (Input)

```
public class testcase1      {
    int    a;
    int    b;
    public int    adder() {
        return 0;
    }
}
```

Figure 9.13: Appendix F - Test Case 1 (Output)

Testing referencing from class to method:

```
public class testcase2
{
    int i = 5;
    int j = 5;

    public int add()
    {
        return i + j;
    }
}
```

Figure 9.14: Appendix F - Test Case 2 (Input)

```
public class testcase2      {
    int    a = 5;
    int    b = 5;
    public int    add() {
        return a+b;
    }
}
```

Figure 9.15: Appendix F - Test Case 2 (Output)

Testing ”this” operator:

```
public class testcase3
{
    int i = 5;

    public int add(int i)
    {
        return i + this.i;
    }
}
```

Figure 9.16: Appendix F - Test Case 3 (Input)

```
public class testcase3      {
    int    a = 5;
    public int    adder(int     b) {
        return b+this.a;
    }
}
```

Figure 9.17: Appendix F - Test Case 3 (Output)

Testing method overloading with predefined variables as arguments:

```
public class testcase4
{
    public int add(int i, int j)
    {
        return i + j;
    }

    public double add(double i, double j)
    {
        return i + j;
    }
}

class testcase4b
{
    testcase4 t = new testcase4();
    int a = 5;
    int b = 5;
    double c = 5.5;
    double d = 4.5;

    int i = t.add(a,b);
    double j = t.add(c,d);
}
```

Figure 9.18: Appendix F - Test Case 4 (Input)

```
public class testcase4      {
    public int    addInt(int     i,int     j) {
        return i+j;
    }
    public double   addDouble(double   i,double    j) {
        return i+j;
    }
}
class testcase4b      {
    testcase4     tc = new testcase4 ();
    int    a = 5;
    int    b = 5;
    double   c = 5.5;
    double   d = 4.5;
    int    i = tc.addInt(a,b);
    double   j = tc.addDouble(c,d);
}
```

Figure 9.19: Appendix F - Test Case 4 (Output)

Testing method overloading with numerical values as arguments:

```
public class testcase5
{
    public int add(int i, int j)
    {
        return i + j;
    }

    public double add(double i, double j)
    {
        return i + j;
    }
}

class testcase5b
{
    testcase5 t = new testcase5();

    int i = t.add(5, 5);
    double j = t.add(5.5, 4.5);
}
```

Figure 9.20: Appendix F - Test Case 5 (Input)

```
public class testcase5alt       {
    public int    addInt(int     i,int     j) {
        return i+j;
    }
    public double   add(double   i,double    j) {
        return i+j;
    }
}
class testcase5b      {
    testcase5alt      t = new testcase5 ();
    int    i = t.add(5,5);
    double   j = t.add(5.5,4.5);
}
```

Figure 9.21: Appendix F - Test Case 5 (Output)

Testing ”this” operator within an inner-class:

```
public class testcase6
{
    int i = 5;

    class testcase6b
    {
        int j = i;
    }

    class testcase6c
    {
        int i = 10;
        int j = this.i;
    }
}
```

Figure 9.22: Appendix F - Test Case 6 (Input)

```
public class testcase6      {
    int    a = 5;
    class testcase6b      {
        int    j = a;
    }
    class testcase6c      {
        int    b = 10;
        int    j = this.b;
    }
}
```

Figure 9.23: Appendix F - Test Case 6 (Output)

Appendix G - Correspondence with Terence Parr

These sections of text are taken from a reply to an email sent to Terence Parr on the 4th of January of 2005. The text remains unchanged however it has been reformatted from the Microsoft Outlook Express email layout into a quote layout more fitting with the report. Not all the text has been presented as only certain sections are particularly relevant to this project.

1. Jonathan Nicholson wrote: ”I see my project basically being a one language translator, it will take a Java file as input, parse it into an AST, it will then be able to modify the tree and then output it.”
   Terence Parr replied with: ”Correct.”

2. Jonathan Nicholson wrote: ”It’s a lot more difficult that I had initially thought from my early research, with a steep but satisfying learning curve.”
   Terence Parr replied with: ”Well, it’s nontrivial ;) There are a few tools built with antlr that do it.”

3. Jonathan Nicholson wrote: ”What do you think of refactoring, and do you commonly practice it?”
   Terence Parr replied with: ”I use it all the time in Intellij’s IDE.”

4. Jonathan Nicholson wrote: ”If you do, do you use software tools such as IntelliJ to do it for you or do you do it by hand?”
   Terence Parr replied with: ”. . . I certainly use IDE based refactoring all the time and feel that the need for constant grooming / refactoring of long-term projects is one of the key lessons I learned building jguru.com.”

Appendix H - Initial work plan

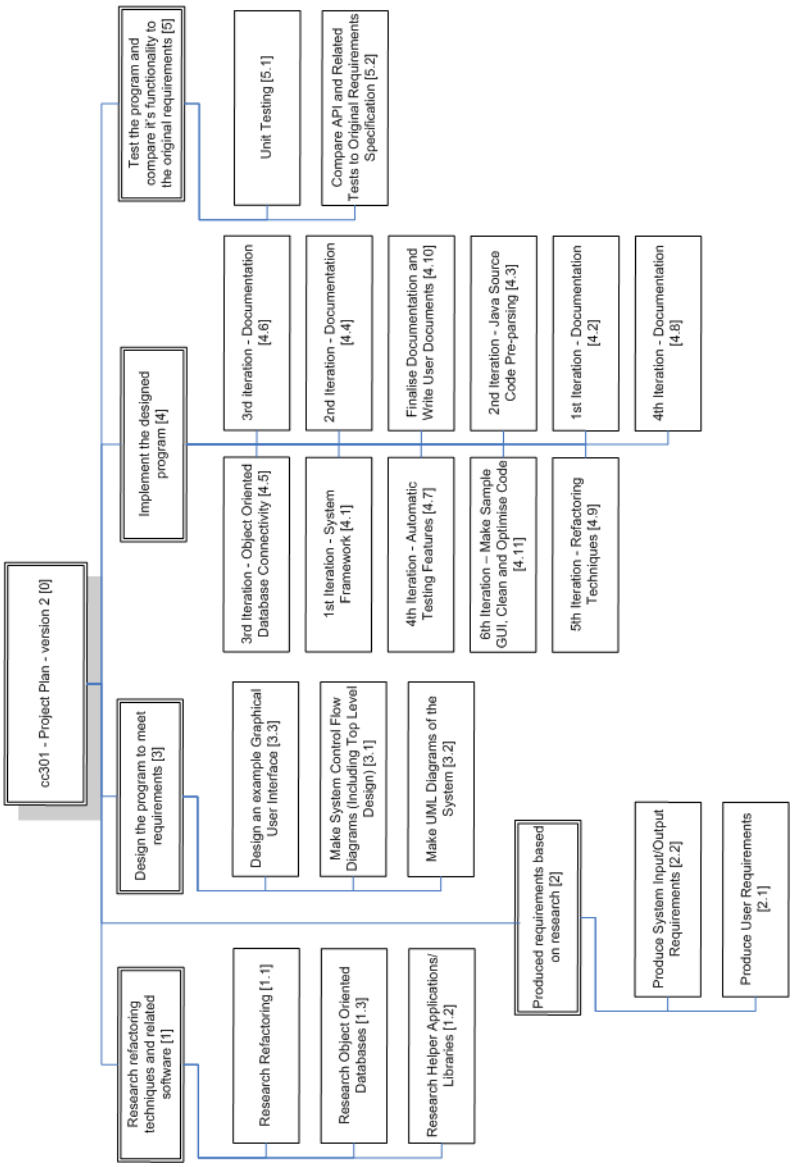
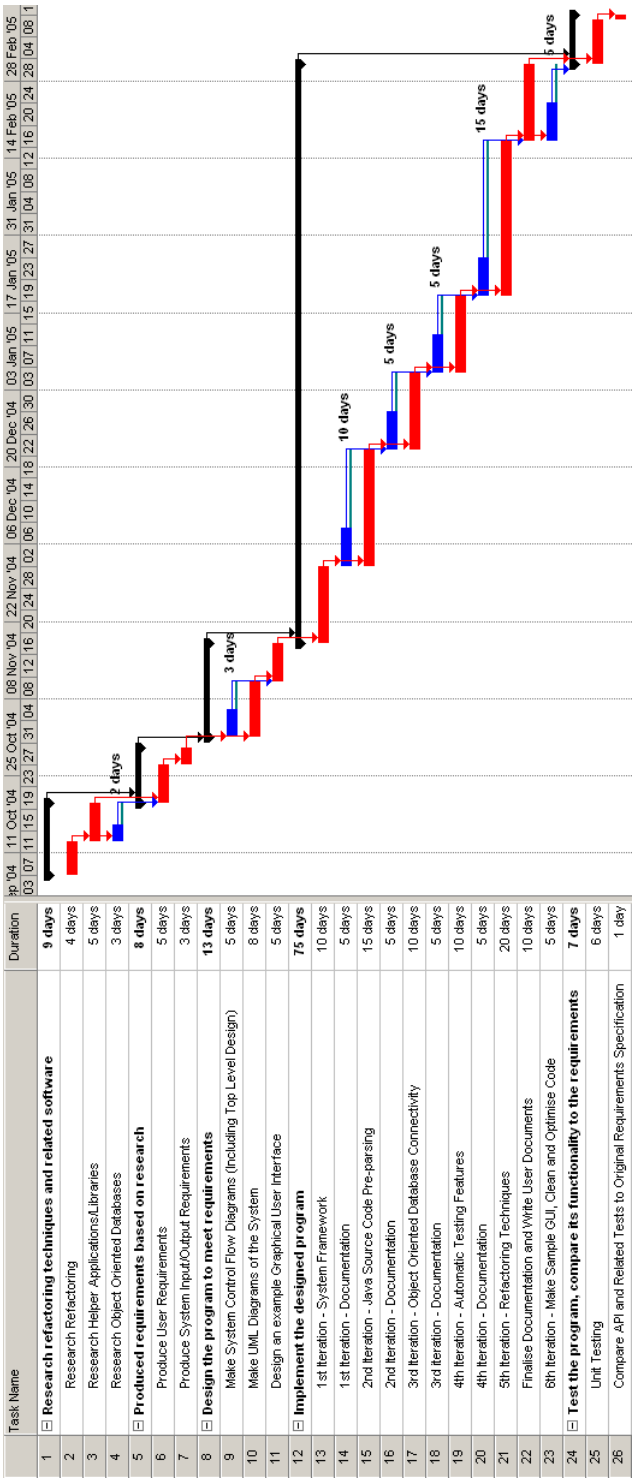


Figure 9.24: Appendix H - Initial work breakdown structure



	Task Name	Duration
1	**Research refactoring techniques and related software**	**9 days**
2	Research Refactoring	4 days
3	Research Helper Applications/Libraries	5 days
4	Research Object Oriented Databases	3 days
5	**Produced requirements based on research**	**8 days**
6	Produce User Requirements	5 days
7	Produce System Input/Output Requirements	3 days
8	**Design the program to meet requirements**	**13 days**
9	Make System Control Flow Diagrams (Including Top Level Design)	5 days
10	Make UML Diagrams of the System	8 days
11	Design an example Graphical User Interface	5 days
12	**Implement the designed program**	**75 days**
13	1st Iteration - System Framework	10 days
14	1st Iteration - Documentation	5 days
15	2nd Iteration - Java Source Code Pre-parsing	15 days
16	2nd Iteration - Documentation	5 days
17	3rd Iteration - Object Oriented Database Connectivity	10 days
18	3rd Iteration - Documentation	5 days
19	4th Iteration - Automatic Testing Features	10 days
20	4th Iteration - Documentation	5 days
21	5th Iteration - Refactoring Techniques	20 days
22	Finalise Documentation and Write User Documents	10 days
23	6th Iteration - Make Sample GUI, Clean and Optimise Code	5 days
24	**Test the program, compare its functionality to the requirements**	**7 days**
25	Unit Testing	6 days
26	Compare API and Related Tests to Original Requirements Specification	1 day

Figure 9.25: Appendix H - Initial schedule