Investigation of a C++ Refactoring Tool

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Ryan Brubaker

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

August, 2008

Investigation of a C++ Refactoring Tool

By Ryan Brubaker

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Dr. Kenny Hunt Examination Committee Chairperson	Date
Dr. Kasi Periyasamy Examination Committee Member	Date
Dr. David Riley Examination Committee Member	Date

ABSTRACT

Brubaker, Ryan, M., “Investigation of a C++ Refactoring Tool”, Master of Software Engineering, August 2008, Advisors: Dr. Kenny Hunt, Dr. Kasi Periyasamy.

The practice of software refactoring has become a core issue in software engineering today. Continually improving the structure of a program, while preserving its observable behavior, extends the lifetime of a program and allows it to evolve to meet ever changing and increasingly demanding requirements. This manuscript describes a prototype for a tool, Automated Refactoring Tool (ART), which assists C++ developers in performing refactorings that improve the structure and readability of their code. The tool provides a C++ preprocessor along with a parser that generates a program database. The developer can then manipulate the program elements within the database to perform refactorings on the source code. The refactoring correctly updates the source code and preprocessing directives to reflect the developer’s intentions and outputs the updated source code to disk. A simple GUI is provided that allows the developer to easily choose with program element to refactor.

ACKNOWLDEGEMENTS

I would like to express my sincere thanks to my project advisors, Dr. Kenny Hunt and Dr. Kasi Periyasamy, for their valuable guidance. Dr. Periyasamy provided valuable input in helping formulate the project idea. Dr. Hunt provided valuable input throughout each project phase, guiding the project to an acceptable conclusion. I especially want to thank Dr. Hunt for the time he spent meeting with me to review project progress. I would also like to thank William Opdyke, Donald Roberts, Alejandra Garrido and Edward Willink for their valuable contributions to the academic literature. Each of their theses was invaluable to me throughout this project. Finally, I would like to thank my wife Tiffany for her patience and encouragement throughout this project. I truly appreciate the sacrifices she made that allowed me to complete this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Abstract Syntax Tree (AST)

An Abstract Syntax Tree is a representation of the syntax of a program's source code, where each node of the tree represents a construct in the grammar of the programming language in which the program was coded.

Directed Graph

A graph in which the pair of vertices representing an edge are ordered.

Extended Backus-Naur Form Grammar (EBNF Grammar)

A notation capable of expressing context-free grammars used to formally describe programming languages.

Integrated Development Environment (IDE)

An Integrated Development Environment is a software application that provides comprehensive software functionality such as source code editing and compilation to computer programmers.

LL Parser

A top-down parser that parses input from left-to-right and constructs a left-most derivation of the input sentence.

Recursive Descent Parser

A top-down parser built from a set of procedures where each procedure implements one of the production rules of the grammar. The structure of the resulting program mirrors that of the grammar it parses.

Visitor Pattern

A software design pattern that separates an algorithm from the structure on which it operates. This allows for the creation of new operations on the structure without modifying the structure itself.

Waterfall Model

A software development model in which development is done in a sequential manner with phases such as requirements analysis, design, implementation and testing.

1. Introduction

In his seminal book, *Refactoring, Improving the Design of Existing Code*, Martin Fowler provides two definitions for the word refactoring:

> ***Refactoring*** *(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*
>
> ***Refactoring*** *(verb): to restructure software by applying a series of refactorings without changing its observable behavior* [1].

Both of these definitions emphasize the two main conditions in evaluating the effectiveness of a refactoring. The first condition is that the refactoring improves a program's design and structure. A change made to optimize a program's performance may require many changes, but might actually make the code harder to understand and thus does not qualify as a refactoring [1]. The second condition ensures that after the refactoring is complete, the program retains its observable behavior and provides the same functionality as it did before the refactoring.

It is a generally accepted fact within the software community that the time spent maintaining software constitutes a large portion of the total cost of software production. It is rarely the case that the initial design of a software product remains unchanged as it is required to handle new customer requirements and is used in ways other than the original intent. Refactoring provides a technique to address both of these concerns. It decreases the cost of software maintenance as it creates an environment where changes to software are easier to make. This environment also allows the program to evolve in ways that were not foreseen and makes the program more robust.

1.1. Related Work

This project drew extensively from three theses originating from the University of Illinois at Urbana-Champaign. This section gives an overview of each of these theses and describers their relevance to this project.

The first major doctoral thesis to address refactoring was William Opdyke's "Refactoring Object-Oriented Frameworks" [2]. Opdyke provided formal definitions for many major refactorings that form the basis for refactoring literature today. His other major contribution was in defining several program properties that must be preserved during refactoring in order to ensure that program behavior does not change. Along with these program properties, he also enumerated the possible program domains that can be affected during a refactoring. Using these properties and domains, he then created formal functions that could be used to verify that necessary preconditions would be satisfied before performing a given refactoring. A violation of one of these preconditions would result in a potential change to program behavior, thus making the refactoring invalid. For example, when changing the name of a class member variable, it is necessary to ensure that the new variable name does not conflict with an existing variable name. This check includes analyzing the names of both inherited class member variables and global variables.

The next major paper in the refactoring literature is Donald Roberts' thesis "Practical Analysis for Refactoring" [3]. Roberts extended Opdyke's previous work by creating the Refactoring Browser for programs written in Smalltalk. This application automated the refactoring process by checking refactoring preconditions and updating the source code to reflect the output of the refactoring. Roberts also defined several criteria that determine both the technical and practical success of an automated refactoring tool.

Finally, Alejandra Garrido's thesis entitled "Program Refactoring in the Presence of Preprocessor Directives" [4], analyzed the difficulties of implementing an automated tool for programs written in C. Her thesis resulted in the implementation of CRefactory, an automated refactoring tool for C. She also contributed several important algorithms used in this project, namely how to process preprocessor directives when refactoring C++ code.

1.2. Project Goals

The original goal of this project was to build on the previous contributions to the refactoring literature by creating a refactoring tool for C++ programs. The program would provide a set of refactorings that would allow C++ developers to quickly and easily modify their code in several ways. The tool would function much like an IDE and allow users to select portions of the code to refactor.

After more research into the project idea, it was decided that such a tool would be beyond the scope of the degree requirements. Therefore, the project goal was refined to implement a prototype of such a tool, verifying that refactorings for C++ were possible and implementing a framework into which future refactorings could easily be developed and tested. Most of the GUI requirements were eliminated to make sure more time was spent on the underlying refactoring functionality.

2. Requirements

The following section provides an overview of the process used to gather requirements for the Automated Refactoring Tool (ART). A waterfall process was used as the software development model throughout the project. Because of this choice, the developer expended significant effort during the requirements phase to review the existing literature and determine what was possible and to understand the work that had previously been completed. Although this approach may have prevented the developer from implementing more functionality in the final product, it did prevent the developer from straying down a wrong path ending in a dead-end.

The original sponsor for this project was Firstlogic, Inc. The project itself was the idea of the developer and was not a request for software from the sponsor. The sponsors at Firstlogic understood that the project was more like a research project and were not concerned with obtaining a final product to be used within the company. Therefore, the developer was free to determine the essential requirements for the project. These requirements were also reviewed and approved by the project supervisor.

In his doctoral thesis [3], Don Roberts provides two categories of criteria that an automated refactoring tool must pass. The requirements for this project attempted to conform to both the technical and practical criteria noted by Roberts and listed below.

- Technical Criteria
    - The tool must maintain a program database that can be searched for various program entities across the entire program.
    - The tool must maintain an Abstract Syntax Tree (AST) to allow the manipulation of the source code.
    - The tool must provide a reasonable assurance that it preserves program behavior.
- Practical Criteria
    - The analysis and transformation of the code must happen in an amount of time that is acceptable to a developer. Otherwise they will just perform the refactoring manually.

- The tool must support an "Undo" function to allow developers to revert a refactoring that does not result in the expected benefits of the refactoring.
- The tool must be integrated within the programmer's Integrated Development Environment (IDE).

For this project, the developer was more concerned with the technical criteria than in providing a practical tool for everyday use. Although "Undo" functionality was included in the original requirements, it had a low priority and was never implemented. Also, ART was always intended to be a stand-alone tool since integration into a third-party IDE was beyond the scope of this project.

A requirements document was generated that listed 48 requirements of which 31 were categorized as functional requirements and 17 categorized as GUI requirements. The document was created solely by the developer after an extensive literature review and received approval from both the project sponsor and the project supervisor. An overview of the requirements document is given in the following section. The original requirements document was not altered after its initial creation.

2.1. Functional Requirements

2.1.1. Refactoring Support Requirements

These requirements were intended to provide lower-level support to the high-level refactorings available to the user. The requirements were as follows:

Requirement #	**Requirement Name**	**Requirement Description**
3.1.1	Find Variable References	Find all references to a variable within the source program.
3.1.2	Find Function Calls	Find all calls to a function within the source program.
3.1.3	Find Variable Name Conflicts	Determine if renaming a variable will result in a name conflict with existing variable declarations.
3.1.4	Save Refactoring	Save the input to a refactoring, which could

	Information	be used to "Undo" the refactoring at a later time.
3.1.5	Undo Refactoring	Revert the changes that were made by a refactoring.
3.1.6	Report Unsatisfied Precondition	Check to make sure that a refactoring's preconditions are satisfied before performing the refactoring.

Table 2.1. Refactoring Support Requirements

The final program did not include requirements 3.1.4 and 3.1.5. Also, although a refactoring's preconditions were checked before performing the refactoring, the preconditions that were checked were not exhaustive.

2.1.2. Implicit Refactoring Requirements

These requirements consisted of smaller refactorings needed to implement the higher-level refactorings available to the user. For instance, to encapsulate a variable, it is necessary to change the access control mode of the variable, create member functions and convert all variable references to access function calls. The requirements were as follows:

Requirement #	**Requirement Name**	**Requirement Description**
3.2.1	Change Access Control Mode	Change the access control mode (e.g. public to private) of a class member variable.
3.2.2	Create Member Function	Create a new member function within a class.
3.2.3	Remove Member Function	Remove an existing member function from a class.
3.2.4	Convert Variable References to Access Function Calls	Used within the Encapsulate Function refactoring. Updates all references to a variable to a call to a *setx* or a *getx* based on whether the reference was a read/write

		reference.
3.2.5	Convert Access Function Calls to Variable References	The inverse of function 3.2.4.
3.2.6	Inline Function Call	Replace a call to a function with the code contained within the function.
3.2.7	Move Member Variable to Subclasses	Moves a member variable declared in a subclass to a declaration in each of its subclasses.

Table 2.2. Implicit Refactoring Requirements

The final program did not include requirements 3.2.3, 3.2.5, 3.2.6 and 3.2.7. Each of these requirements was needed to implement "Undo" functionality for some of the higher-level refactorings. Since "Undo" functionality was not implemented, these requirements were no longer necessary.

2.1.3. Explicit Refactoring Requirements

These requirements were the planned refactorings that would be automated and available to a user of ART. The following refactorings were chosen for implementation:

Requirement #	**Requirement Name**	**Requirement Description**
3.3.1	Encapsulate Variable	Make a public variable private, provide public access functions to the variable and update all references to the variable with calls to the access functions.
3.3.2	Rename Variable	Rename a variable, updating its declaration and all of its references with the new variable name.
3.3.3	Rename Member Function	Rename a class member function, updating its declaration and all calls to the function

		with the new function name.
3.3.4	Extract Function	Extract a portion of code into its own function and replace it with a call to the function.
3.3.5	Decompose Conditional	Extract a complex conditional expression into its own function and replace it with a call to the function.
3.3.6	Move Member Variable to Superclass	Replace duplicate declarations of a variable within multiple subclasses, with one variable declaration in their common superclass.

Table 2.3. Explicit Refactoring Requirements

The final program included the first three requirements from this table. After consultation with the project sponsor and supervisor, the remaining refactorings were omitted from the final implementation.

2.1.4. Project Requirements

These requirements were intended to cover the concept of a "project" within the ART program and were as follows:

Requirement #	**Requirement Name**	**Requirement Description**
3.4.1	Extract Source File Information	The original requirement was to extract source file information from a Microsoft Visual Studio™ project file.
3.3.2	Extract Include Path Information	The original requirement was to extract include path information from a Microsoft Visual Studio™ project file.

Table 2.4. Project Requirements

The final program did not work with Microsoft Visual Studio™ programs. The first requirement was satisfied by finding all source files underneath a given file system directory. At this time, the program does not work with any include path information.

2.1.5. Preprocessor Requirements

These requirements covered all of the functionality dealing with the preprocessor needed for an automated refactoring program and were as follows:

Requirement #	**Requirement Name**	**Requirement Description**
3.5.1	Process Include Directives	The preprocessor must handle #include directives appropriately in the context of an automated refactoring program.
3.5.2	Process Macro Definitions	The preprocessor must handle macro definitions and expansions appropriately in the context of an automated refactoring program.
3.5.3	Process Conditional Directives	The preprocessor must handle conditional directives appropriately in the context of an automated refactoring program.
3.5.4	Preserve White Space	The preprocessor must preserve the original white space of a program after a refactoring occurs.
3.5.5	Preserve Comments	The preprocessor must preserve the comments in a program after a refactoring occurs.

Table 2.5. Preprocessor Requirements

Each of the requirements in this section were at least partially fulfilled in the final program. See Section 3.1 for further details.

2.1.6. Parser Requirements

These requirements covered all of the functionality dealing with the C++ parser needed for an automated refactoring tool and were as follows:

Requirement #	Requirement Name	Requirement Description
3.6.1	Report Parsing Errors	The parser must report any parsing errors that occur
3.6.2	Parse Complete C++ Programs/Libraries	The parser should be able to parse any library or executable program written in C++.
3.6.3	Parse Code Fragments	This requirement was intended to make sure the user selected a set of valid C++ statements when performing the *Extract Function* refactoring.
3.6.4	Save Parse Information	This requirement was intended to save parse information between program sessions to save work reparsing a program.
3.6.5	Output Program	Output a parsed program to disk.

Table 2.6. Parser Requirements

2.2. Technical Difficulties and Risk Analysis

The first major obstacle in developing an automated refactoring tool for C++ programs is the presence of a preprocessor. Preprocessing directives provide three major challenges that must be handled with functionality that differs from typical preprocessing behavior:

- *#include* directives must not destroy the modularity a programmer depends on when separating source code into separate files.
- Conditional directives (e.g. *#ifdef*, *#ifndef*) must not eliminate code that may break when program elements from other parts of the program are refactored. It is also important to be able to reproduce all branches of a conditional directive in order to reproduce the original program that existed before preprocessing occurred.

- Macro definitions must be updated to reflect refactorings that occur on program elements used in the macro definitions. Also, macros may have multiple definitions when defined in different conditional directives. These definitions must be preserved for the same reason as the second bullet point listed above.

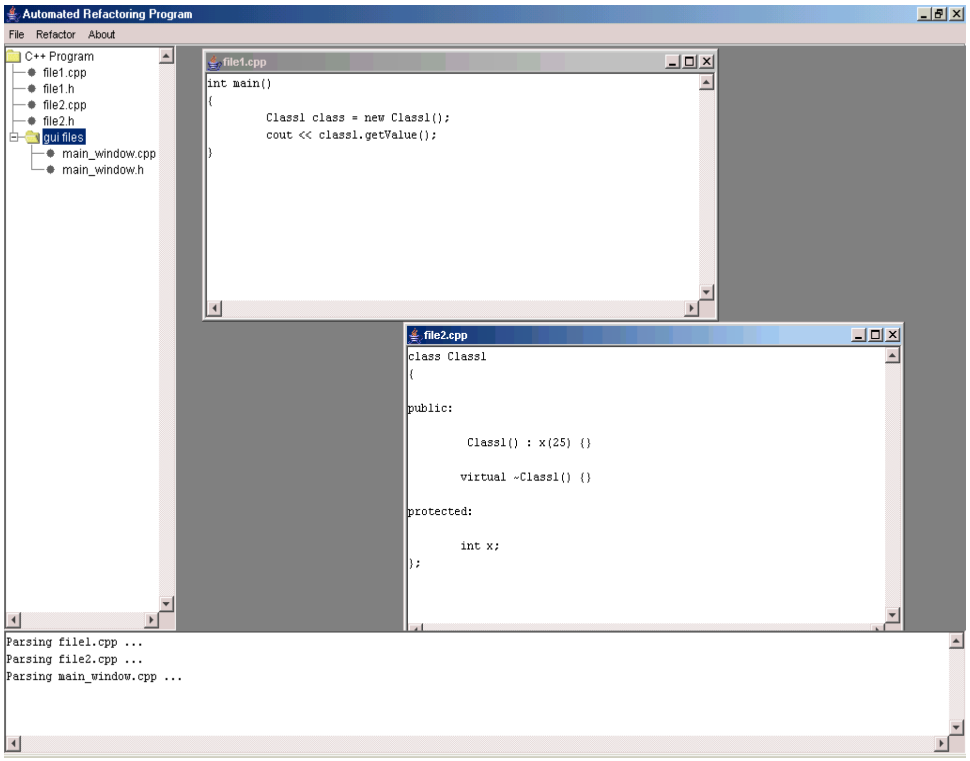
These issues and their potential solutions are described in much greater detail in Section 3.1.

The next major obstacle in development of such a tool is the requirement to implement a parser for the C++ language. C++ has a very large and complex grammar, made even more difficult to parse because semantic information is required to make a correct parse. The website at [5] provides a good overview of the difficulties of parsing C++ and provides links to several approaches that have been taken over the years.

Finally, the complexity of the C++ language makes it difficult to ensure a refactoring can preserve the behavior of a program. For instance, “Move Member Variable to Superclass” is a common refactoring that consolidates a duplicate declaration of a variable within two subclasses to one declaration in a common parent class. However, this refactoring would change the memory footprint of any object that had a type of those affected by the refactoring. Although it would be considered bad programming style, a developer could access a member variable of such an object by pointer arithmetic. Because the offsets of the member variables may have been affected by the refactoring, the refactoring may have broken expected behavior and introduced a bug in the program [1].

2.3. GUI Requirements

The GUI requirements were developed in order to provide a user of ART with a simple interface that would make it easy to refactor portions of a source program. In general, the GUI was intended to look like a typical IDE without the ability to edit source-code “documents.” Instead, the user would be able to select portions of a source program to refactor.



```
int main()
{
        Class1 class = new Class1();
        cout << class1.getValue();
}
```

```
class Class1
{

public:

         Class1() : x(25) {}

        virtual ~Class1() {}

protected:

        int x;
};
```

```
Parsing file1.cpp ...
Parsing file2.cpp ...
Parsing main_window.cpp ...
```

Figure 2.1. Original GUI Prototype

The GUI Requirements were broken into categories based on each component contained within the GUI.

GUI Component	**Requirement Description**
Main Window	The Main Window provided a container to hold all of the other GUI components.
Menu Bar	The Menu Bar was intended to provide menus for the user to initiate much of the functionality available in the program.
File Menu	The File Menu was intended to provide the following menu items: • "Open Project…" would open a project file for the ART program. • "Close Project" would close the current open project.

	• "Save" would save any refactorings that had been performed and output the modified source program. • "Exit" would close the program.
Refactor Menu	The Refactor Menu was intended to provide the following menu items, each of which would perform the corresponding refactoring based on the code selected by the user: • "Encapsulate Variable…" • "Rename Variable…" • "Rename Member Function…" • "Extract Function…" • "Decompose Conditional…" • "Move Member Variable to Superclass…" • "Undo Refactoring…"
About Menu	The About Menu was intended to provide a single menu item to provide information about the ART program.
File Navigator Panel	The File Navigator Panel was intended to display the source files contained in the source program and allow the user to open these files.
Output Panel	The Output Panel was intended to provide status/output messages for actions that occurred (e.g. parsing, parsing errors) in the ART program.
Display Panel	The Display Panel was intended to display source file contents and allow the user to select portions of code to refactor.

Table 2.7. GUI Requirements

3. Design

This section provides a brief overview of the design of each ART program component. More details are given in Section 5.

3.1. Preprocessor

After some initial research, the developer decided to use the Wave library [6] provided within the Boost Framework [7]. The Boost Wave library is a Standards conformant, and highly configurable implementation of the mandated C99/C++ preprocessor functionality hidden behind an easy to use iterator interface [8]. This library was essential to the project as it provided a solid foundation for preprocessing functionality. Without this library, the developer may have been required to build a complete preprocessor from scratch, which would have been beyond the scope of the project. The developer was able to modify and extend the Wave library to implement much of the functionality needed for a preprocessor within the context of an automated refactoring tool.

Figure 3.1. shows the initial UML class diagram that served as the design for the preprocessor.

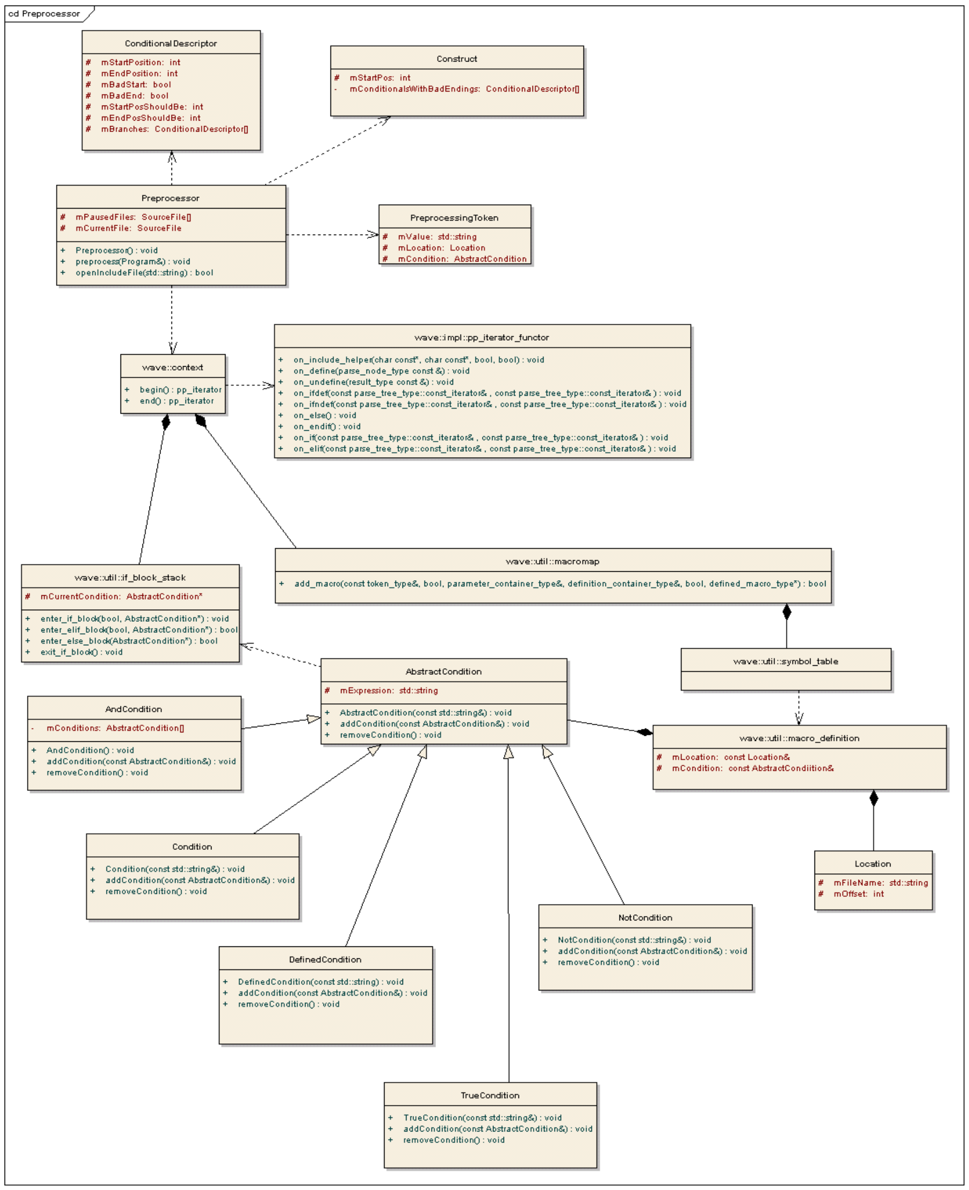

cd Preprocessor
ConditionalDescriptor
# mStartPosition: int
# mEndPosition: int
# mBadStart: bool
# mBadEnd: bool
# mStartPosShouldBe: int
# mEndPosShouldBe: int
# mBranches: ConditionalDescriptor[]
Construct
# mStartPos: int
- mConditionalsWithBadEndings: ConditionalDescriptor[]
Preprocessor
# mPausedFiles: SourceFile[]
# mCurrentFile: SourceFile
+ Preprocessor() : void
+ preprocess(Program&) : void
+ openIncludeFile(std::string) : bool
PreprocessingToken
# mValue: std::string
# mLocation: Location
# mCondition: AbstractCondition
wave::context
+ begin() : pp_iterator
+ end() : pp_iterator
wave::impl::pp_iterator_functor
+ on_include_helper(char const*, char const*, bool, bool) : void
+ on_define(parse_node_type const &) : void
+ on_undefine(result_type const &) : void
+ on_ifdef(const parse_tree_type::const_iterator& , const parse_tree_type::const_iterator&) : void
+ on_ifndef(const parse_tree_type::const_iterator& , const parse_tree_type::const_iterator&) : void
+ on_else() : void
+ on_endif() : void
+ on_if(const parse_tree_type::const_iterator& , const parse_tree_type::const_iterator&) : void
+ on_elif(const parse_tree_type::const_iterator& , const parse_tree_type::const_iterator&) : void
wave::util::if_block_stack
# mCurrentCondition: AbstractCondition*
+ enter_if_block(bool, AbstractCondition*) : void
+ enter_elif_block(bool, AbstractCondition*) : bool
+ enter_else_block(AbstractCondition*) : bool
+ exit_if_block() : void
wave::util::macromap
+ add_macro(const token_type&, bool, parameter_container_type&, definition_container_type&, bool, defined_macro_type*) : bool
wave::util::symbol_table
AbstractCondition
# mExpression: std::string
+ AbstractCondition(const std::string&) : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void
AndCondition
- mConditions: AbstractCondition[]
+ AndCondition() : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void
wave::util::macro_definition
# mLocation: const Location&
# mCondition: const AbstractCondiition&
Condition
+ Condition(const std::string&) : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void
Location
# mFileName: std::string
# mOffset: int
NotCondition
+ NotCondition(const std::string&) : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void
DefinedCondition
+ DefinedCondition(const std::string) : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void
TrueCondition
+ TrueCondition(const std::string&) : void
+ addCondition(const AbstractCondition&) : void
+ removeCondition() : void

Figure 3.1. UML Class Diagram for a Preprocessor

The Preprocessor class provided the main functionality for the overall preprocessing. At construction, the Preprocessor takes an object of type Program (explained in Section 3.2.). The Preprocessor iterates over all of the source files that are contained in the Program object, preprocessing each source file and creating a list of preprocessing tokens for the source file. Internally, the Preprocessor class uses a context class provided by the Wave library to preprocess each source file.

The diagram also shows several classes, ConditionalDescriptor and AbstractCondition and its descendents, that are used to provide special preprocessing functionality needed for a refactoring tool. The purpose of these classes is discussed further in Section 5.

The classes in this diagram with a “wave::util::” prefix were classes that existed in the Wave preprocessor and were reused by the developer. However, the developer did have to update the internal functionality of these classes to implement the special preprocessing functionality needed in the context of an automated refactoring tool (see Section 5).

3.2. Parser

Given the complexity of C++, an open source or third party parsing library was sought for use on this project. The Spirit parser, part of the Boost Framework was considered but not adopted due to the anticipated steep learning curve. The well known Lex/Yacc system was also considered but rejected since it was unclear whether it could fully support the needs of this project to maintaining whitespace significance and construct explicit abstract syntax trees for dynamic manipulation. The decision was eventually made to write a customized recursive descent parser.

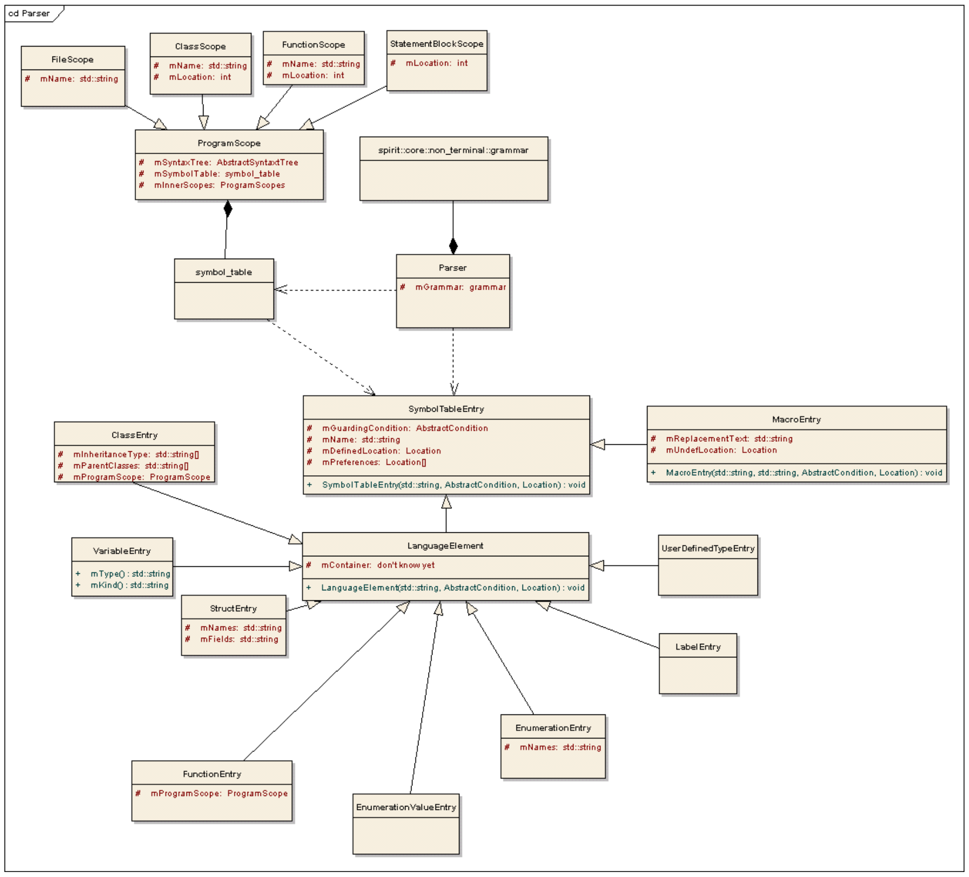

cd Parser
FileScope
# mName: std::string
ClassScope
# mName: std::string
# mLocation: int
FunctionScope
# mName: std::string
# mLocation: int
StatementBlockScope
# mLocation: int
ProgramScope
# mSyntaxTree: AbstractSyntaxTree
# mSymbolTable: symbol_table
# mInnerScopes: ProgramScopes
spirit::core::non_terminal::grammar
symbol_table
Parser
# mGrammar: grammar
SymbolTableEntry
# mGuardingCondition: AbstractCondition
# mName: std::string
# mDefinedLocation: Location
# mPreferences: Location[]
+ SymbolTableEntry(std::string, AbstractCondition, Location) : void
MacroEntry
# mReplacementText: std::string
# mUndefLocation: Location
+ MacroEntry(std::string, std::string, AbstractCondition, Location) : void
ClassEntry
# mInheritanceType: std::string[]
# mParentClasses: std::string[]
# mProgramScope: ProgramScope
LanguageElement
# mContainer: don't know yet
+ LanguageElement(std::string, AbstractCondition, Location) : void
VariableEntry
+ mType() : std::string
+ mKind() : std::string
UserDefinedTypeEntry
StructEntry
# mNames: std::string
# mFields: std::string
LabelEntry
EnumerationEntry
# mNames: std::string
FunctionEntry
# mProgramScope: ProgramScope
EnumerationValueEntry

Figure 3.2. UML Class Diagram for a Parser

After careful consideration, the developer decided against using the Spirit parser as it would have required much effort to gain a very good understanding of the library, a task that would have taken a great deal of time. In the end, the developer decided to write a recursive descent parser as this technique is straight-forward and did not require a large learning curve. To implement the parser, the developer created classes for each of the grammar rules and grammar constructs that were implemented in ART. A Parser class exists that defines a "parse*X*" function for each of the implemented grammar constructs. Each function parses the particular corresponding construct of the grammar. The developer utilized the extended C++ grammar developed by Edward Willink in his thesis "Meta-Compilation for C++" [10]. The use of this grammar instead of the official C++ grammar resulted in a simplified parser and eliminated the need for contextual information while parsing.

Many of the concepts from the UML diagram in Figure 3.2 were still used during the implementation of ART. A ProgramDatabase class was designed that stores all of the program elements defined in the program. These elements are contained within a Scope class, which is used to delineate each scope defined in the program. The existence of scopes allows the ART program to do some basic contextual processing to ensure that the source program does not have duplicate declarations of a variable within a scope. It also allows the ART program to ensure that refactorings affecting a variable at a certain scope, do not affect variables with the same name defined in a different scope.

To handle the source files contained in the source program, the following UML class diagram was created:

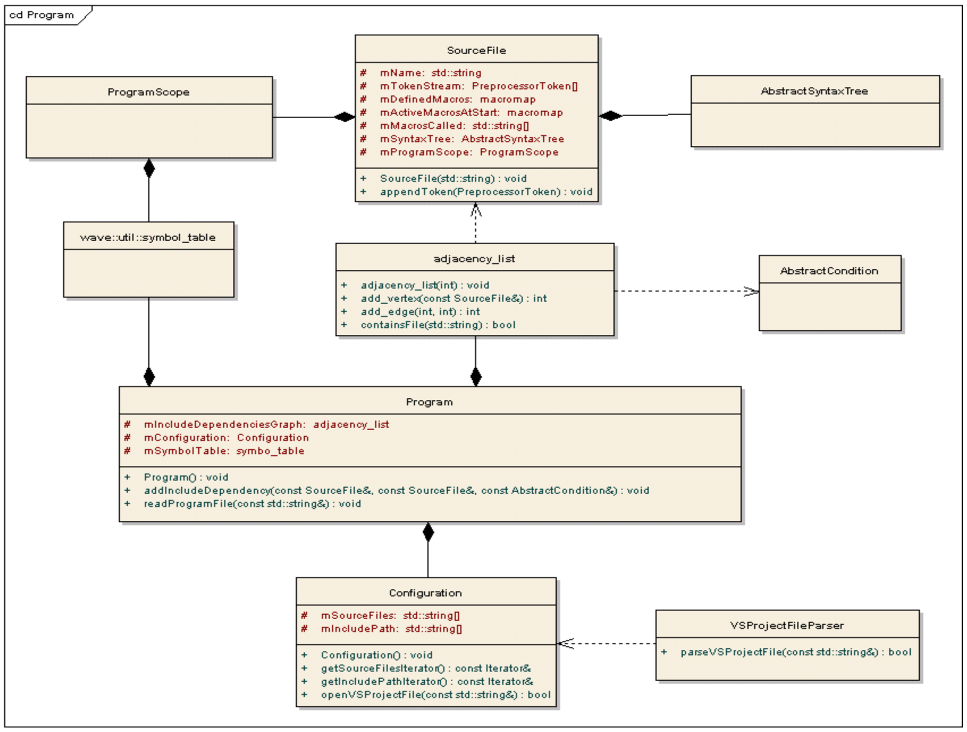

cd Program
SourceFile
# mName: std::string
# mTokenStream: PreprocessorToken[]
# mDefinedMacros: macromap
# mActiveMacrosAtStart: macromap
# mMacrosCalled: std::string[]
# mSyntaxTree: AbstractSyntaxTree
# mProgramScope: ProgramScope
+ SourceFile(std::string) : void
+ appendToken(PreprocessorToken) : void
ProgramScope
AbstractSyntaxTree
wave::util::symbol_table
adjacency_list
+ adjacency_list(int) : void
+ add_vertex(const SourceFile&) : int
+ add_edge(int, int) : int
+ containsFile(std::string) : bool
AbstractCondition
Program
# mIncludeDependenciesGraph: adjacency_list
# mConfiguration: Configuration
# mSymbolTable: symbo_table
+ Program() : void
+ addIncludeDependency(const SourceFile&, const SourceFile&, const AbstractCondition&) : void
+ readProgramFile(const std::string&) : void
Configuration
# mSourceFiles: std::string[]
# mIncludePath: std::string[]
+ Configuration() : void
+ getSourceFilesIterator() : const Iterator&
+ getIncludePathIterator() : const Iterator&
+ openVSProjectFile(const std::string&) : bool
VSProjectFileParser
+ parseVSProjectFile(const std::string&) : bool

Figure 3.3. UML Class Diagram for a Program

The Program class holds a list of the source files within the source program that is being refactored, an include dependencies graph for the source program and the program database for the source program. Each file is represented by a SourceFile class that contains both the file's list of preprocessing tokens and the file's AST that is formed during parsing.

3.3. Refactoring Engine

The initial design of the refactoring engine resulted in the following UML class diagram:

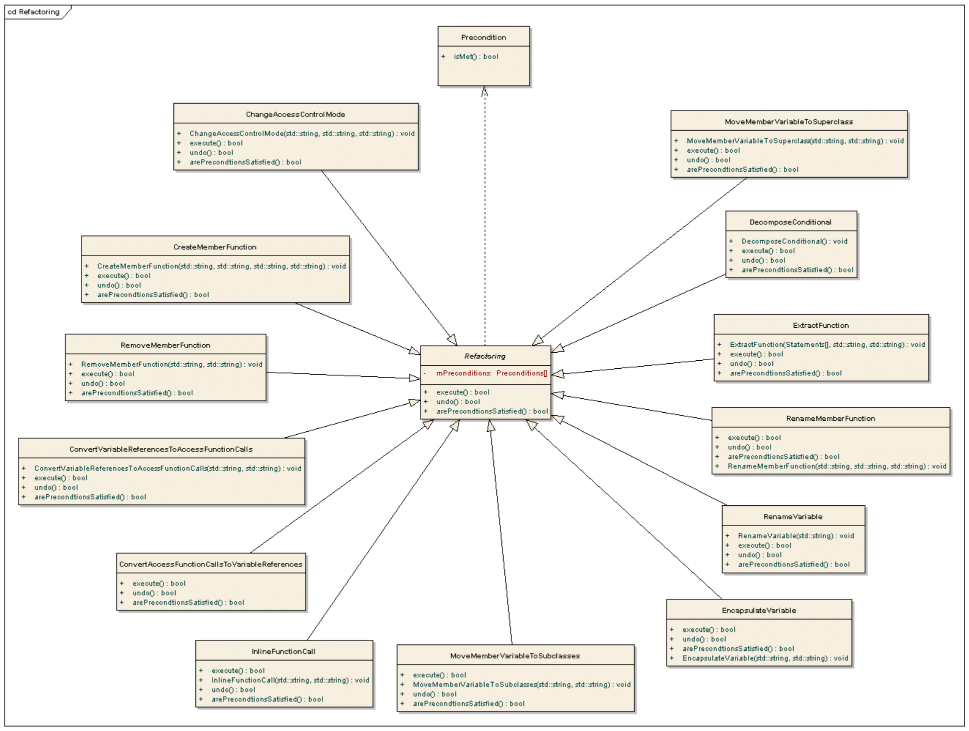

cd Refactoring
Precondition
+ isMet() : bool
ChangeAccessControlMode
+ ChangeAccessControlMode(std::string, std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
MoveMemberVariableToSuperclass
+ MoveMemberVariableToSuperclass(std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
DecomposeConditional
+ DecomposeConditional() : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
CreateMemberFunction
+ CreateMemberFunction(std::string, std::string, std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
ExtractFunction
+ ExtractFunction(Statements[], std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
RemoveMemberFunction
+ RemoveMemberFunction(std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
Refactoring
- mPreconditions: Preconditions[]
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
RenameMemberFunction
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
+ RenameMemberFunction(std::string, std::string, std::string) : void
ConvertVariableReferencesToAccessFunctionCalls
+ ConvertVariableReferencesToAccessFunctionCalls(std::string, std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
RenameVariable
+ RenameVariable(std::string) : void
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
ConvertAccessFunctionCallsToVariableReferences
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
EncapsulateVariable
+ execute() : bool
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
+ EncapsulateVariable(std::string, std::string) : void
InlineFunctionCall
+ execute() : bool
+ InlineFunctionCall(std::string, std::string) : void
+ undo() : bool
+ arePrecondtionsSatisfied() : bool
MoveMemberVariableToSubclasses
+ execute() : bool
+ MoveMemberVariableToSubclasses(std::string, std::string) : void
+ undo() : bool
+ arePrecondtionsSatisfied() : bool

Figure 3.4. UML Class Diagram for a Refactoring Engine

Each refactoring was intended to have its own class that would perform the necessary modifications on the source program and output the code.

However, the refactoring engine was designed using several classes that implement the Visitor Pattern [11]. There are a total of five concrete classes along with a base class that implement this functionality. The base ASTVisitor class defines a "visit*X*" function for each of the implemented grammar constructs. Each function visits a node in the AST that corresponds to the particular construct of the grammar, performing any actions needed to implement the refactoring engine. Table 3.1. lists each visitor class and its responsibility:

ASTProgramDbVisitor	This class visits each node of the AST, creating entries in the program

	database for each program element defined in the program.
ASTRenameVariableVisitor	This class visits each node of the AST, renaming any references of a variable that is being renamed.
ASTEncapsulateVariablePrinterVisitor	This class is responsible for creating the get/set functions created during an “Encapsulate Variable” refactoring.
ASTEncapsulateVariableReferenceUpdaterVisitor	This class is responsible for updating each reference to a variable being encapsulated with the appropriate get/set function.
ASTPrinterVisitor	This class visits each node printing out the terminal tokens to a file and is used to output a program after a refactoring occurs.

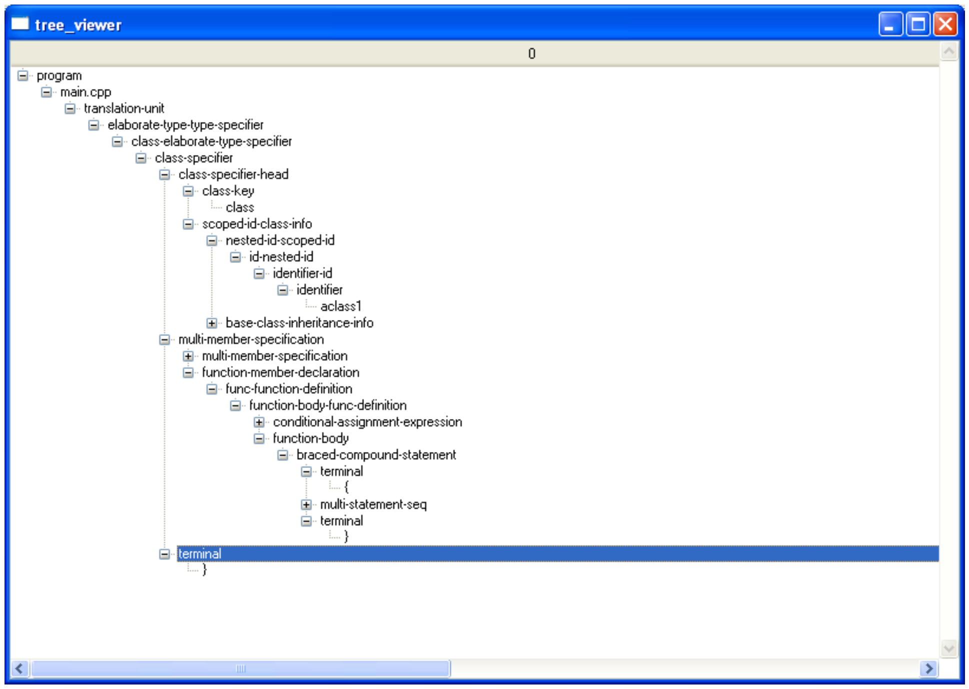
Table 3.1. Visitor Classes Used to Implement a Refactoring Engine

4. Implementation and Testing

The implementation and testing phase consisted of transforming the requirements and design into code. The developer first focused on extending the Wave preprocessor to implement the functionality needed for a preprocessor within the context of an automated refactoring system. In retrospect, the developer spent too much time in this phase, which negatively affected the time spent on more important functionality such as the parser and refactoring engine. This mistake limited the amount of functionality in the final tool.

Testing was utilized throughout the project and was automated as much as possible. Using the unit testing framework provided by the Boost Framework, the developer implemented eleven different test suites to verify the correctness of the program. These tests were mainly black box tests used in testing the parser and preprocessor. For the preprocessor, each test would take in a source file and verify the correct output of the preprocessor by checking the tokens it produced. For the parser, each test would take in a source file and verify the state of the program database that was created. These tests also served as regression tests to ensure that changes to the code did not break existing functionality.

The developer also created a small test application that displayed a parse tree formed by the parser functionality in ART. This application helped the developer to diagnose problems within the parser. A screenshot of this application appears below:



```
program
  main.cpp
    translation-unit
      elaborate-type-type-specifier
        class-elaborate-type-specifier
          class-specifier
            class-specifier-head
              class-key
                class
              scoped-id-class-info
                nested-id-scoped-id
                  id-nested-id
                    identifier-id
                      identifier
                        aclass1
                base-class-inheritance-info
            multi-member-specification
              multi-member-specification
              function-member-declaration
                func-function-definition
                  function-body-func-definition
                    conditional-assignment-expression
                    function-body
                      braced-compound-statement
                        terminal
                          {
                        multi-statement-seq
                        terminal
                          }
            terminal
              }
```

Figure 4.1. AST Display Application

5. Description of Refactoring with ART

This section provides a detailed description of the three main components of the ART application.

5.1. Preprocessor

As previously mentioned, the developer used the Wave preprocessor as the starting point for the preprocessor and extended its functionality to make it suitable for an automated refactoring tool.

5.1.1. Conditional Directives

Conditional directives introduce significant complexity when developing a automated refactoring tool. The normal behavior of a preprocessor is to process only those branches that fall within conditional branches that evaluate to true. Since, however, the observable behavior of the source code must remain unchanged regardless of the target platform, refactoring must take into account all branches within a conditional directive.

The first problem occurs when a conditional branch evaluates to false causing the preprocessor to ignore the program tokens. In this example from [4], a conditional directive is used to provide two different definitions for a typedef declaration:

```
#if __STDC__
typedef void* pointer;
#else
typedef char* pointer;
#endif
```

Figure 5.1. Sample of Multiple Declarations of a Program Element

Assuming the program was being processed with the ***__STDC__*** symbol defined, the normal preprocessing behavior would eliminate the second declaration. However, if such behavior was preserved in a refactoring tool and the user performed a rename refactoring on the remaining declaration, the declaration that was eliminated would remain unchanged. If the program was then compiled with the ***__STDC__*** symbol undefined, compiler errors would result everywhere *pointer* was updated with the new name used in the refactoring.

A refactoring tool must also preserve all conditional branches to preserve the contents of the original program after a refactoring occurs. If the preprocessor ignores the tokens of conditional branches that evaluate to false, these portions of the program would be lost when the program is written back to disk.

The solution, outlined in [4], is to process every conditional branch as if it evaluates to true. To implement this functionality, it is first necessary to label each preprocessing token with a label that represents the current condition in the program. Using the code in Figure 5.1. as an example, the tokens in the first typedef declaration would be labeled with the logical condition *__STDC__*. The tokens in the second typedef declaration would be labeled with the logical condition *!__STDC__*, where the "!" symbol stands for the logical negation operator. In this way, both declarations can exist within the same program since they are differentiated by the conditional label that is attached to them.

To correctly label preprocessing tokens, the preprocessor must implement a "Current Condition Stack" [4]. As the preprocessor processes program tokens, it maintains a stack that keeps track of the current program condition. In the program example above, the preprocessor would take the following actions:

- When processing the line "`#if __STDC__`", the preprocessor would push the logical condition *__STDC__* onto the top of the stack.
- When processing the line "`#else`", the preprocessor would pop the top of the stack and push the logical condition *!__STDC__* onto the top of the stack.
- When processing the line "`#endif`", the preprocessor would pop the top of the stack leaving it empty.

Each token that the preprocessor processes is labeled with the condition that exists on the top of the stack.

Processing all branches of a preprocessing conditional directive allows a refactoring tool to handle multiple declarations of program elements. However, it also introduces the possibility of a new problem. Conditional directives allow for the possibility of incomplete syntactic units within a conditional branch. The code in Figure

5.2. shows an example where processing each branch of code would result in a parsing error since the tokens "`for (`" are only available for the first branch of the conditional:

```
for (
#if BY_ROW
     i=0; i<R; i++)
     s+=a[i];
#elif BY_COL
     j=0; j<C; j++)
     s+=a[j];
#endif
```

Figure 5.2. Sample of an Incomplete Syntactic Construct

The solution to this problem, provided in [4], is to implement a "Conditional Completion Algorithm." This algorithm updates the token stream so that each conditional branch contains a complete syntactical construct. Complete syntactic constructs were defined as the following C grammar constructs in [4]:

- Statement
- Declaration
- Structure field
- Enumerator field
- Array initializer value.

When processing the code in Figure 5.2., the Conditional Completion Algorithm would modify the source code to that of Figure 5.3.

```
#if BY_ROW
     for (i=0; i<R; i++)
          s+=a[i];
#elif BY_COL
     for (j=0; j<C; j++)
          s+=a[j];
#endif
```

Figure 5.3. Sample of a Fixed Incomplete Syntactic Construct

By adding the "`for (`" tokens to each branch, the preprocessor ensures that each branch contains a complete syntactical construct as defined by the C++ grammar and will not cause a parsing error.

To implement the Conditional Completion Algorithm, the preprocessor works in two passes through the source program [4]. The first pass inserts tokens into the token stream to mark where incomplete syntactic constructs occur. The second pass through the code reorganizes program tokens based on the previously inserted markers to ensure that each branch contains a complete syntactical construct.

To determine the existence of incomplete syntactic constructs, the preprocessor keeps track of its state with regard to parsing the constructs listed above. For example, while parsing the *for* statement in Figure 5.3., the preprocessor is aware that its current state at the point of the *#if* directive is not a valid state for a preprocessing directive to occur. The preprocessor then inserts a special token into the source program marking the beginning of an invalid preprocessing conditional directive that will need to be fixed during the second pass of the preprocessor.

To implement the state awareness functionality, the developer used a hash map that mapped certain key tokens (e.g. *for*, *enum*, etc…) to a structure that contained the following items:

- The state to transition to after processing the token.
- A function that served as a precondition that had to be checked before transitioning to the new state.
- A vector of function pointers representing the actions to take to transition to the new state.

For example, the *for* token has the following values in its mapped structure:

- The value *IN_FOR* representing the new state of parsing a for statement.
- A function representing a *TRUE* condition as a *for* token always results in a transition to the *IN_FOR* state.
- A list of two function pointers that do the following:
    - Push the *IN_FOR* state onto the top of the state stack.
    - Reset a variable that tracks the number of open parentheses for the current *for* statement.

After the first pass of the preprocessor is complete, the preprocessor rescans the token stream looking for the special tokens that signal a bad conditional directive. Each of the marker tokens contains information about the construct that enables the preprocessor to correctly modify the token stream. The information consists of the following:

- A flag indicating the preprocessing conditional has a bad start.
- A flag indicating the preprocessing conditional has a bad ending.
- The start position of the preprocessing conditional directive.
- The end position of the preprocessing conditional directive.
- The position at which the start of the conditional should be.
- The position at which the end of the conditional should be.
- A list of tokens that need to be inserted at the start of the conditional branches in order to make them complete.
- A list of tokens that need to be inserted at the end of the conditional branches in order to make them complete.

This information allows the conditional completion algorithm to rearrange the program tokens to ensure that each branch of a conditional directive contains a complete syntactic construct from the constructs listed above**.**

5.1.2. Macros

To handle macro definitions within an automated refactoring tool, it is necessary to implement special functionality for both macro definitions and macro expansion.

Macro definitions are stored in a macro definition table. A normal macro table would only allow one definition for a macro at any given time. However, because the ART must process all branches of preprocessing conditional directives, the macro definition table must be able to handle multiple definitions of the same macro.

```
#ifdef LARGE_CLASS_SIZES
     #define CLASS_SIZE = 50
#else
     #define CLASS_SIZE = 25
#endif
```

Figure 5.3. Example of a Macro with Multiple Definitions

In the case of Figure 5.3., the macro definition table must store both of the definitions for *CLASS_SIZE*. This behavior is possible by differentiating each definition by the logical condition label that was previously explained in Section 5.1.1.

Because the macro definition table can hold multiple definitions for a given macro, it becomes necessary to expand a macro call for all of its possible definitions. Using the macro defined in Figure 5.3. as an example, a call to the macro such as the following:

```
int classSize = CLASS_SIZE
```

Figure 5.4. Call to a Macro With Multiple Definitions

would result in the following code being generated by the preprocessor:

```
int classSize =
#ifdef LARGE_CLASS_SIZES
     50
#else
     25
#endif
;
```

Figure 5.5. Expansion of a Macro With Multiple Definitions

This expansion results in an incomplete conditional directive, which the Conditional Completion Algorithm would fix later on in the preprocessing process.

It is also necessary to check whether a refactoring on a program element requires a change within a macro body. If so, the automation functionality must verify that the macro is not called in a context where the program element has a different definition. Figure 5.6, taken from [4], shows an example of this situation:

```
#define ER1 errstatus = 1
int f1() {
     int errstatus
     …
     if (bottom < 0)
          ER1;
     …
}

int main() {
     int errstatus;
     …
     if (input == 0)
          ER1;
     …
}
```

Figure 5.6. Example of a Macro Called from Two Different Contexts

At each call to the macro *ER1*, there is a local variable named *errStatus*, resulting in two different contexts for the use of this variable. If *errStatus* is renamed in function *f1* the use of *errStatus* in the macro definition must also be updated with the new variable name. However, that would then result in a compiler error in the *main* function as the macro expansion for *ER1* would result in an undefined variable. Therefore, if a refactoring is applied to a program element that affects a macro definition, it is necessary to examine all other calls to that macro and ensure there are no other scopes in which the macro is called with a different definition for the program element [4]. Otherwise, the refactoring cannot be performed safely and must be canceled. The developer was not able to include this functionality in the final program.

5.1.3. Include Directives

When processing include directives, a normal C++ preprocessor does not make any effort to preserve the modularity created by the programmer in his use of different source files. Instead, the preprocessor processes include files as a continuation of the current file, resulting in the different files merged into one long token stream. An automated refactoring tool however, needs to preserve the modularity of separate source files in order to present the source code to the user in a readable format. Therefore, the preprocessor within ART maintains objects for each source file, each of which contain its program tokens. The file objects are stored within a directed graph, where each directed edge represents an include dependency. With this representation, it is possible for the preprocessor to reuse the processed tokens of a source file if it has previously been included from another source file.

Although the developer implemented functionality to preserve modularity and create an included dependency graph, no formal testing was done to verify this functionality in the final program.

5.2. Parser

The parser within the ART is implemented using the familiar technique of a recursive-descent parser [12]. A separate class exists for each construct in the C++ grammar. The parser has a *parseX* function that creates an object for each grammar construct it encounters in the program.

Instead of creating a parser based off of the official C++ grammar, the developer decided to use the FOG grammar specified in [10]. The FOG grammar is a superset of C++ and was developed to eliminate the need for contextual information while parsing C++ programs. This made the implementation of the parser much easier as syntactic processing and contextual processing did not have to be combined into one step.

After the program is parsed, the ART creates a program database that contains all of the elements (e.g. classes, variables, functions, etc…) declared in the program. This database is created through the use of a class that implements the Visitor Pattern [12]. The visitor travels down the nodes of the AST, creating an entry in the program database for each program element contained in the tree. The visitor also keeps track of references to the variables and functions the program declares, by storing the AST nodes that represent the statements and expressions that contain the references to the program elements. Finally, the visitor also performs some minimal contextual analysis such as reporting multiple declarations of a variable that exist within the same scope or reporting a reference to an undeclared variable. The program database is then used to allow users to refactor the program elements it contains.

5.3. Refactoring Engine

The refactoring engine is also implemented through several classes them implement the Visitor pattern.

5.3.1 Rename Variable/Function

The rename variable/function refactoring is used to rename a variable element within a program and update all references to that variable with the new name. With the following program listed in Figure 5.7 the ART displays the GUI shown in Figure 5.8:

```
class aclass1
{
private:
     int a;

public:

     int get_a() const
     {
          return a;
     }

     void set_a(int in_a)
     {
          a = in_a;
     }

     int b;
     bool c;
     bool d;

     int test()
     {
          set_a( 5 );
     }
}
```

Figure 5.7. Sample Program for Rename Variable Refactoring

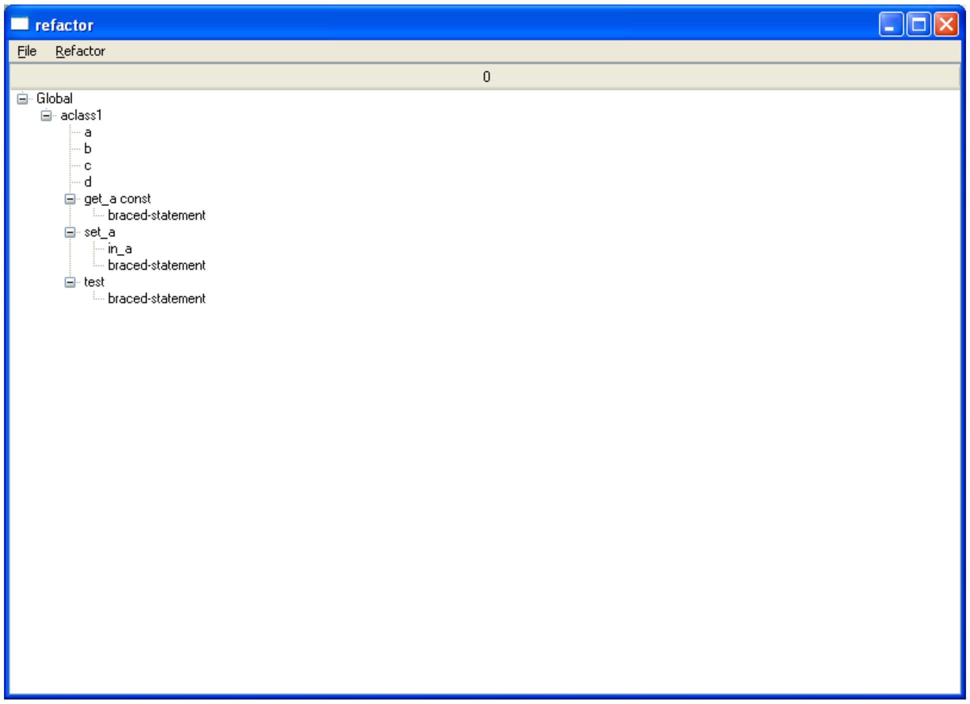


Figure 5.8. GUI Display for Rename Variable Refactoring Sample Program

The user can then select a program element to rename:

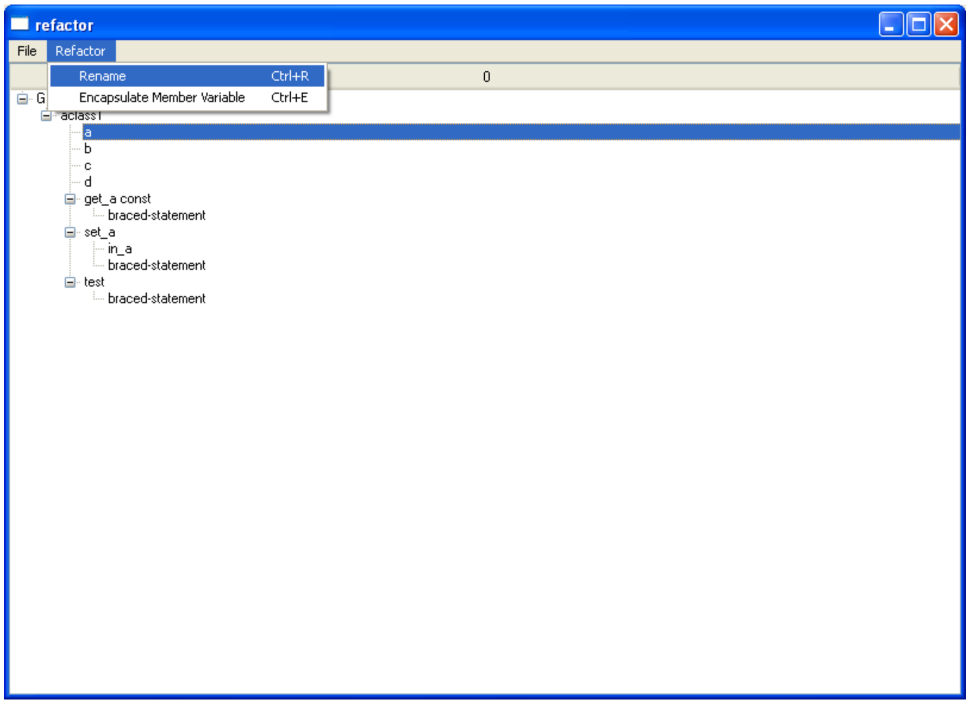
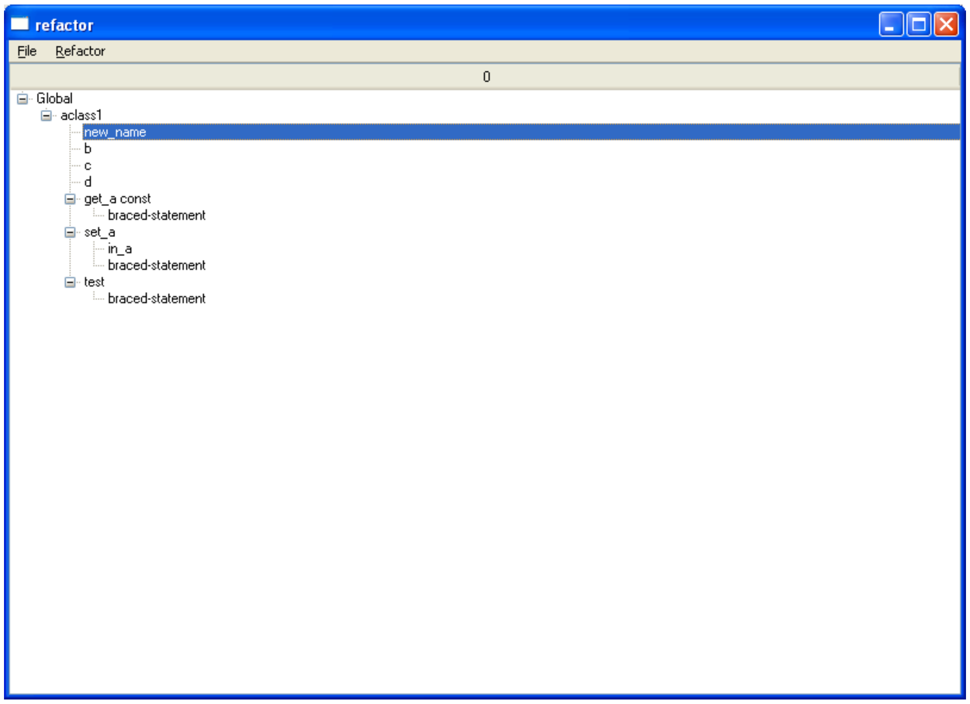


Figure 5.9. Renaming a Variable

After entering a new name, *new_name*, for the variable *a*, the source code is updated to that of Figure 5.10 while the GUI is also updated to reflect the change (Figure 5.11).

```
class aclass1 {

private:

     int new_name;

public:

     int get_a() const
     {
          return new_name;
     }

     void set_a(int in_a)

     {
          new_name = in_a;
     }

public:
     int b;
     bool c;
     bool d;

     int test()
     {
          set_a( 5 );
     }
}
```

Figure 5.10. Source Code Results for Rename Variable Refactoring



```
Global
  aclass1
    new_name
    b
    c
    d
    get_a const
      braced-statement
    set_a
      in_a
      braced-statement
    test
      braced-statement
```

Figure 5.11. GUI Results for Rename Variable Refactoring

The rename refactoring can rename class variables, function parameters, local variables and class member functions. If the user selects a name that is already used within the scope of the program element, the ART will report an error.

As noted earlier, the ART keeps track of references to variables and functions that the program declares when creating the program database. To implement the rename functionality, the ART uses a Visitor pattern that visits the statements and expressions that contain the references to the variable or function. When the visitor reaches the AST node representing the program element reference, it updates the node with the new name chosen by the user. The ART then uses another Visitor that rewrites the updated source code back to disk.

5.3.2 Encapsulate Variable Refactoring

The Encapsulate Variable refactoring makes a public variable private, creates get/set functions for the variable and updates all variable references with the appropriate function. With the following program listed in Figure 5.12 the ART displays the GUI shown in figure 5.13:

```
class aclass1 {

public:
     int a;
     int b;
     bool c;
     bool d;

     void testFunction1()
     {
          int z = a;
          z += 3;
     }

     void testFunction2()
     {
          if (b == 3)
          {
               b = a;
          }
          else
          {
               a = 3;
          }
     }
}
```

Figure 5.12. Sample Program for Encapsulate Variable Refactoring

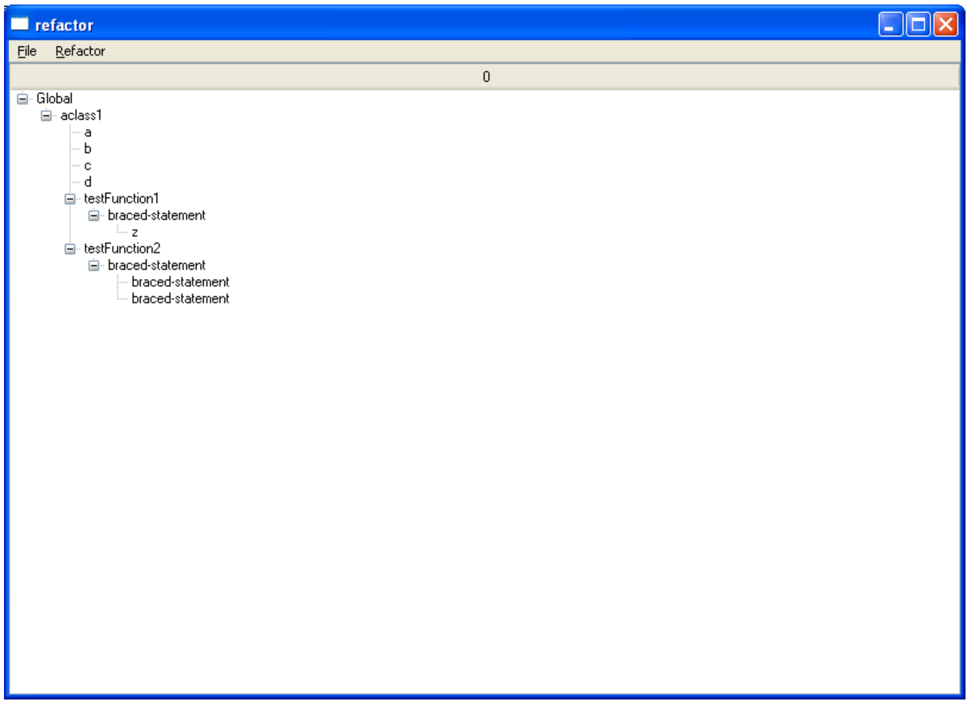
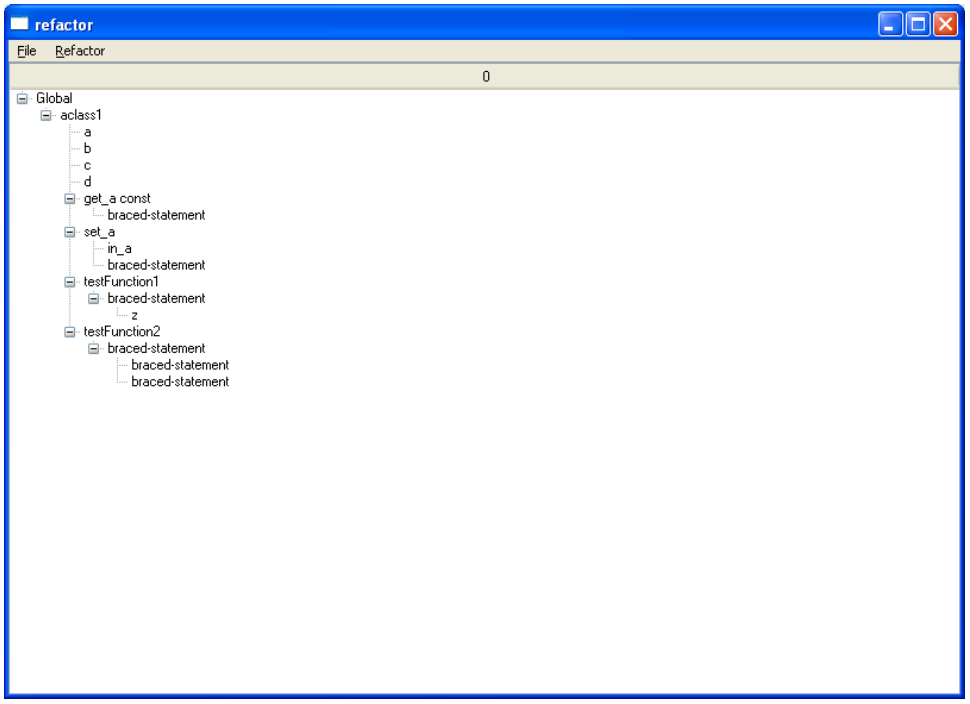


Figure 5.13. GUI Display for Encapsulate Variable Refactoring

If the user then selects to encapsulate the variable *a*, the source code is updated to that of Figure 5.14, while the GUI is also updated to reflect the change (Figure 5.15).

```
class aclass1 {

private:

     int a;

public:

     int get_a() const
     {
          return a;
     }

     void set_a(int in_a)
     {
          a = in_a;
     }

public:
     int b;
     bool c;
     bool d;

     void testFunction1()

     {
          int z = get_a();
          z = 3;
     }

     void testFunction2()
     {
          if (b == 3)
          {
               b = get_a();
          }
          else
          {
               set_a( 3 );
          }
     }
}
```
Figure 5.14. Source Code Results for Encapsulate Variable Refactoring



```
Global
  aclass1
    a
    b
    c
    d
    get_a const
      braced-statement
    set_a
      in_a
      braced-statement
    testFunction1
      braced-statement
        z
    testFunction2
      braced-statement
        braced-statement
        braced-statement
```

Figure 5.15. GUI Results for Encapsulate Variable Refactoring

The ART will only allow a user to encapsulate a public variable. If the user attempts to perform the Encapsulate Variable refactoring on a protected or private variable, the program will report an error.

To implement this functionality, the ART first uses a Visitor pattern that updates all of the variable references to the appropriate get/set function. During program database creation, a reference to a variable is marked with a flag indicating whether the reference is a read-only reference, or if the reference is assigned to. In this way, the visitor class knows which function to use during the update.

The ART then uses another Visitor pattern that rewrites the program back to disk. During this operation, the visitor writes out the new declaration of the variable along with the definitions of the get/set functions. It also skips over the previous declaration of the variable to remove that declaration from the program. The ART then rereads the updated program to load all of the changes made by the refactoring.

6. Limitations

The ART provides a good basis for an automated refactoring tool for C++ programs. However, there are several limitations to this tool before it has the potential to become useful to C++ developers.

The preprocessor used in ART implements only a subset of the possible cases that arise during the implementation of the Conditional Completion Algorithm1. These cases as defined in [4] as the following:

- A conditional with a bad start.
- A conditional with a bad ending.
- A conditional with a bad start and a bad ending.
- Two conditionals with the same logical condition that break the same syntactic construct.
- Two conditionals, the second one breaking an inner statement (e.g. the first conditional breaks the expression in a 'while' statement and the second condition breaks the statements within the 'while' loop.
- Two overlapping conditionals, the first not having any inner conditionals.
- Two overlapping conditionals, the first having inner conditionals.

The preprocessor developed in ART is able to handle the first two cases. Although some of the latter cases would likely be very rare in real code, they must be handled to ensure correctness of programs. Also, although the preprocessor creates an include dependencies graph, the functionality to handle include directives has not been implemented in any meaningful way. The ART needs to incorporate #include directives as part of the C++ grammar and provide functionality to open and parse included files. Finally, the macro processing functionality does not check all contexts in which macros are called and would not prevent a refactoring in the case where a program element has a different definition in another context within the program.

Of all of the program components in ART, the parser functionality has the most limitations. As stated previously, parsing C++ is a difficult task because of the complexity of the language. Even though the parser handles 114 of the C++ grammar rules specified in [10], these rules do not cover the entire grammar. The grammar rules that were implemented include the following C++ constructs:

- Class definitions
- Member variable declarations
- Member function definitions
- Assignment statements
- Assignment expressions(This includes expressions that use arithmetic operators such as '+' and "*' and also logical operators such as '&&' and '||'.
- All major control statements
    - if/else statements
    - for loops
    - while loops
    - do/while loops
    - switch statement

The parser does not implement some of the more complex features of C++, which include the following:

- templates
- the '->' and '.' operators
- memory management (e.g. "new" and "delete")
- constructors and destructors
- exceptions

The refactoring engine did not implement all of the refactorings specified in the original requirements. "Extract Function," "Decompose Conditional," and "Move Member Variable to Superclass" were omitted due to their complexity and a lack of time at the end of the project. The current GUI also would not allow for certain refactorings such as Extract Function, since it does not display the code to the user.

7. Continuing Work

There are several areas where ART could be expanded if further work is desired. Though much work has been done with the preprocessor, the Conditional Completion Algorithm did not implement all of the cases presented in [4]. Although the Include Dependency Graph is created, no real testing has been done to process multiple source files and make sure refactoring works across source files. There is also more work that could be done to ensure that refactorings do not break macro definitions and calls.

The largest area for more work is the parser. More research would be needed into the FOG grammar developed in [10] to make sure it is a suitable substitute for the official C++ grammar within a refactoring tool. Another possibility might be to look into the use of parser generation tools such as lex/yacc to see if they could be utilized to ease the creation of a parser. However, it would be necessary to verify such tools would be able to preserve white space and comments that existed in the program.

More refactorings could be included in the refactoring engine along with more checking of preconditions for the existing refactorings. It would also be necessary to update the GUI to be more like the original idea of an IDE to allow for refactorings that require the selection of actual source code from the program.

8. Conclusion

This manuscript describes the basic features of an automated refactoring tool for programs written in C++. The ART provides initial implementations for much of the major functionality that would be needed in a complete refactoring tool. The work done by others in the field of refactoring has also provided important information that must be included in any tool that supports automated refactoring capabilities.

Although the limitations specified in Section 4 are large enough to prevent the ART from being a useful tool at this point, this manuscript has shown that a useful automated refactoring tool for C++ programs is possible. Further work on the preprocessor and the parser contained in ART would provide a solid foundation to allow more refactorings to be implemented. Adding more functionality to ensure that the refactoring tool preserves program behavior, would assure developers that program use is safe and results in better code.

Ultimately, for the ART to be useful, it would need to be integrated into an existing C++ IDE. It is highly unlikely that a C++ developer would settle for a tool that must run separately from the program in which most of his work occurs. Therefore, more work is needed to explore how the ART could be integrated into other tools that already create their own program databases and abstract syntax trees.

9. Bibliography

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley, 1999.

[2] W. Opdyke, "Refactoring Object-Oriented Frameworks", Department of Computer Science, Diss. University of Illinois at Urbana-Champaign, 1992.

[3] D. Roberts, "Practical Analysis for Refactoring", Department of Computer Science, Diss. University of Illinois at Urbana-Champaign, 1999.

[4] A. Garrido, "Program Refactoring in the Presence of Preprocessor Directives", Department of Computer Science, Diss. University of Illinois at Urbana-Champaign, 2005.

[5] http://www.nobugs.org/developer/parsingcpp, "Parsing C++", 2001

[6] http://www.boost.org/doc/libs/1_35_0/libs/wave/index.html, "Wave V1.3"

[7] http://www.boost.org/, "Boost C++ Libraries"

[8] http://www.boost.org/doc/libs/, "Boost C++ Libraries"

[9] http://www.boost.org/doc/libs/1_35_0/libs/spirit/index.html, "Spirit User's Guide"

[10] E. Willink, "Meta-Compilation for C++", Diss. University of Surrey, 2001.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[12] D. Watt, D. Brown, *Programming Language Processors in Java: Compilers and Interpreters*, Prentice Hall, 2000.