International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:2, 2015

Designing a Tool for Software Maintenance

Amir Ngah, Masita Abdul Jalil, Zailani Abdullah

Abstract—The aim of software maintenance is to maintain the software system in accordance with advancement in software and hardware technology. One of the early works on software maintenance is to extract information at higher level of abstraction. In this paper, we present the process of how to design an information extraction tool for software maintenance. The tool can extract the basic information from old programs such as about variables, based classes, derived classes, objects of classes, and functions. The tool have two main parts; the lexical analyzer module that can read the input file character by character, and the searching module which users can get the basic information from the existing programs. We implemented this tool for a patterned sub-C++ language as an input file.

Keywords—Extraction tool, software maintenance, reverse engineering, C++.

I. Introduction

SOFTWARE engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [1]. It is also defined as a systematic approach to the analysis, design, assessment, implementation, testing, maintenance and reengineering of software [2]. The generic activities in all software processes are requirement analysis, design, coding, testing and maintenance [4], [3].

Software maintenance refers to the modifications of software after delivery. Other terms suggested for maintenance are *software support*, *software renovation*, *continuation engineering* and *software evolution* [5]. The IEEE Standard 1219-1993 [6] has defined software maintenance as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. ISO/IEC 14764-1999 [7] has defined software maintenance as software product that undergoes modification code and associated documentation due to a problem or the need for improvement. Rajlich [8] has proposed a new direction of software evolution and maintenance. Software systems change and evolve over time. It is impossible to develop any software which does not need to be modified. Therefore, a research and development of CASE tools in software maintenance is very significant.

Maintainers are usually under pressure to accomplish maintenance tasks as quickly as possible. The problem for most maintainers is that they have to maintain unfamiliar code that has been modified and the accompanying documentation is usually out of date, inadequate, inconsistent or sometimes non-existent. Sometimes, the sheer complexity of the programs can make the modification tasks look impossible. More than often, the source code may be the only source of information maintainers have at hand. In theory, the source code itself should contain all the information a maintainer may need to commence a modification. The problem is how the maintainers find a systematic way to uncover this information. One of the early work on building maintenance tools is to gather the basic information from the existing program like information related to variables; local and global, classes, objects of classes, and soon. In this paper, we present the process of how to design an information extraction tool for software maintenance that can collect a basic information from a sub-C++ programs.

Amir Ngah, Masita A. Jalil and Zailani Abdullah are with the School of Informatics and Applied Mathematics, University Malaysia Terengganu, 21030 Kuala Terengganu, Malaysia (e-mail: amirnma@umt.edu.my).

II. Related Works

There are a number of tools developed for software maintenance especially in reverse engineering (e.g. [9], [10], [11], [12], [13]). Risi and Scanniello [9] have developed a visualization tool for the reverse engineering of object-oriented software called MetricAttitude. They considered a number of object-oriented metrics such as weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class, flow info, number of message sends, number of methods, lines of code and number of comments.

Maras et al. [10] have developed phpModeler, a tool for reverse engineering of legacy php web applications that generates static UML diagrams showing resources which the current web page is using, its functions and dependencies it has on other web pages. phpModeler has two main modules: model generator module that generates static UML diagrams representing resources the current web page is using, its functions, properties and dependencies on other web pages. The second module is the difference analyzer that shows differences between web page model versions.

Cseri et al. [11] have developed a software maintenance tool that focuses on the code comprehension process of large legacy C++ systems that heavily utilize code comments. Their research proposed a method to find the correct place of the comments in the AST-based on project-specific rules. Chen et al. [12] have developed a software maintenance tool called a Powerful Live Updating System (POLUS). The tool is capable of iteratively evolving running software into newer versions. POLUS is designed to support realistic software changes involving both code and data. The tool is composed of three components. A patch constructor, in the form of a source to source compiler, detects the semantic differences between two successive software versions and generates the POLUS patch files. A patch injector is a running process that applies the updates. A runtime library provides some utility functions to manage POLUS patches for the patch injector. These all tools are purposely developed to aid software maintenance activities.

III. Sub C++ Program Elements

The C++ language is becoming very popular among programmers and most of the critical systems developed were using this language. The language also provides flexibility for users. We implemented this tool for a sub-C++ language as a input file. The sub-C++ is a subset of the original C++ language that only focuses on some part of elements of the C++ language.

Generally, the vocabulary in a programming language is categorized into four: reserved words, special symbols, numerical, and identifier. The reserved words in sub-C++ are shown in Table I. A few additional words as shown in Table II is also categorized as reserved words in sub-C++. There are two categories of special symbols which are symbol with double character and single character as shown in Table III and Table IV respectively. A numerical in sub-C++ is a decimal notation for a non-negative integer only. For example 22, 0 and 321. An identifier started with a letter, which may be followed by more letters or numeral. For example token, x, and B2.

Basically, all the operations used in the sub-C++ language are true if they are of the same type. In this case, the name of a variable must be unique in the sub-C++ program if and only if the variable name defined in different scopes. Every function in sub-C++ has its own scope. If a programmer defines a local variable in one function and at the same time defines the same name in another function, the definition is correct. However if the name is defined as a global variable; the name cannot be used again in the program even in different scopes.

TABLE I:
RESERVED WORDS in SUB-C++

do	if	for	int	
else	case	char	void	
break	class	switch	return	
public	default	private	continue	protected

TABLE II:
ADDITIONAL RESERVED WORDS in SUB-C++

cin	cout	main
scanf	printf	include

Identifiers and reserved words in sub-C++ are case sensitive, meaning that the words in upper and lower cases are different. Therefore, the words for reserved words must be written in small letters only. If the reserved words were written in

TABLE III:
SPECIAL SYMBOLS with DOUBLE CHARACTER in SUB-C++

<=	>=	==	!=	/*	*/	//

TABLE IV:
SPECIAL SYMBOLS with SINGLE CHARACTER in SUB-C++

’	-	*	/	=	&	<	>	
(	)	{	}	”	’	,	;	.

upper case, the tool will generate error message. However, the identifiers are correct if they have upper and lower case letters. However, the identifiers in upper case letters are different with identifiers in lower case letters. Also, the space between symbols is omitted. For example, the following two statements (S1 and S2) which were written in sub-C++ language are the same. The statement S1 contains more than one space between symbols, and statement S2 does not have any space between the contents.

```
Statement (S1): if ( x > 10 ) x = x  1 ;
Statement (S2): if(x>10)x=x-1;
```

Every line in sub-C++ language must end with a semicolon (;). If the line does not end with a semicolon, it is an incorrect line or fault. For comments, they begin with the special symbol /* and end with the other special symbol */. If the comment does not end with the right symbol, the tool gives an error message. Comments also can start with special symbol //. This comment type is just used for single line comment only.

In sub-C++, an identifier can be a local or global variable, the name of based or derived class, class attribute, method or object. However, sub-C++ only consider a variable type of int and char. A variable can be declared with or without initialization.

IV. A Tool Design

This information extraction tool consists of two main modules. There are the lexical analyzer module, and the searching module as shown in Fig. 1. The lexical analyzer module will analyze the lexical from the input file, sub-C++. The searching module enables users to gather the information from tokens that were produced by the lexical analyzer module. For example, users can get the information like global and local variables, the number of classes, objects of related classes, and so on. The design for each module will be executed separately, where each module is depended on one another.

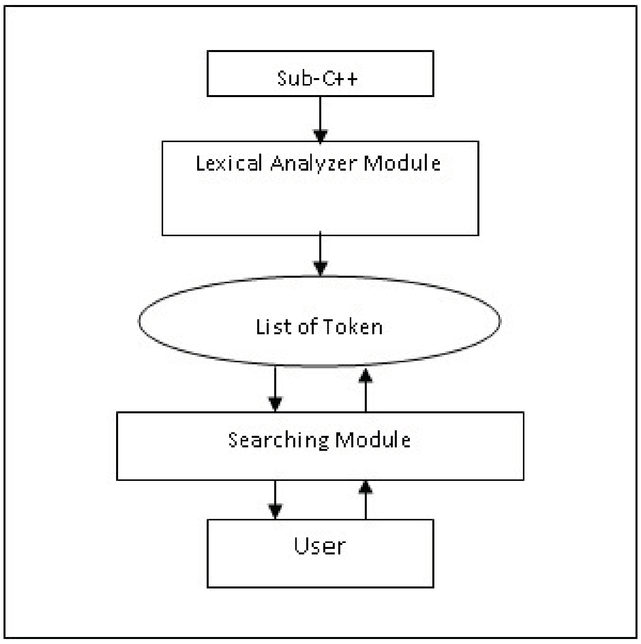

Fig. 1: The Tool Model

A. *The Design of Lexical Analyzer Module*

The first phase in this tool is the lexical analyzer or also known as a scanner, similar to a compiler design. The lexical analyzer reads the sub-C++ program language character by character until it reaches the end of the program. In the process of reading, if the lexical analyzer finds any symbols or characters that does not belong to the sub-C++ language components, it will tell that an error occur. In other words, the lexical analyzer recognizes the basic tokens in its input and represents the tokens in some encoded form. Then, the lexical analyzer transfers it to the next part of the tool, the searching module.

The coding method is used by the lexical analyzer to encode tokens by using an integer value, that is an enumerated type in C++ language. Then, the lexical analyzer uses these values to identify the tokens and passes them to the syntax analyzer. The enumerated type introduces a name for every symbol in the sub-C++ language, as shown in Table V. When the tool recognizes any of the sub-C++ language symbol such as the equal (=) as an input, the lexical analyzer outputs the corresponding symbol value *equal*. It uses the save1() function to save the output values to the output file as shown in (1).

$$save1(equal); \quad (1)$$

In addition, the lexical analyzer uses different output functions to save another kind of tokens, called long tokens, which include the numerical and the value of the numerical constant; the name and the values are the code of identifiers, and the newline value is the line number. The save2() function

TABLE V:
REPRESENTATION of TOKENS USING INTEGER VALUE

name	symbol	integer value
endtxt		0
numeral		1
unknown		2
name		3
newline		4
(Special symbols)		
semicolon	;	5
comma	,	6
not	!	7
signnum	#	8
colon	:	9
doublecolon	::	10
equal	==	11
less	<	12
dot	.	13
greater	>	14
gteq	>=	15
lesseq	<=	16
noteq	!=	17
andpercent	&	18
and	&&	19
or	||	20
plus	+	21
minus	-	22
multiple	*	23
division	/	24
assignment	=	25
cinoperator	>>	26
coutoperator	<<	27
leftparenthesis	(	28
rightparenthesis	)	29
Openbracket	[	30
Closebracket	]	31
Ocompountstate	{	32
Ccompountstate	}	33
(Reserved words)		
cin		34
cout		35
main		36
scanf		37
printf		38
include		39
do		40
if		41
for		42
int		43
else		44
case		45
char		46
void		47
break		48
class		49
switch		50
return		51
public		52
default		53
private		54
continue		55
protected		56

saves tokens such as numerical and followed by its value. For example, the lexical analyzer outputs a variable name as a symbol named *name* followed by the value of that variable as shown in (2).

$$save2(name, value); \tag{2}$$

The line number of the source program is also included in the lexical analyzer. This inclusion helps the tool to provide more informative error messages. When the lexical analyzer reads a new line, it outputs the newline followed by the line number. Also, when the lexical analyzer reads unknown symbols, it saves them as an unknown to the output file. These will later be reported as an error in the next part of the tool. Finally, when the lexical analyzer reaches the end of the file, it outputs *endtxt* and terminates.

Generally, every character in sub-C++ will be classified by lexical analyzer into six (6) groups which are numerical, identifier, reserved word, special symbol, commented line and quotation statement (string). The lexical analyzer will identify numerical characters using the algorithm as shown in Fig. 2.

```
if (isdigit(ch))
{
    int value = 0;
    while (isdigit(ch))
    {
        int digit = ch - '0';
        if (value <= (maxint - digit)/1)
        {
            value = value * 10 + digit;
            ch = RichEdit1->Lines->
               Text.c_str()[character++];
        }
        else
        {
            ListBox1->Items->Add("error");
            while (isdigit(ch))
            ch = RichEdit1->Lines->
               Text.c_str()[character++];
        }
    }
    ListBox1->Items->Add(numeral1);
    ListBox1->Items->Add(value);
}
```

Fig. 2: Algorithm for Numerical Identification

When the lexical analyzer detects letter characters, it will assume that the token may be an identifier or a reserved word. A valid identifier in sub-C++ is the word which is created by combination of letters and numbers. The word must starts with a letter. A single letter is also classified as an identifier. The algorithm to detects identifier is shown in Fig. 3.

The lexical analyzer is also capable of detecting special symbol in sub-C++. There are two types of special symbol; single and double characters. Fig. 4 shows an example of an algorithm to detect a single and double character special symbol. For example, if the lexical analyzer detects an assignment symbol (=), the tool will check the second

```
    if (isalpha(ch))
    {
        int length = 0;
        do
        {
            temptxt[length] = ch;
            ch = RichEdit1->Lines->
                Text.c_str()[character++];
            ++length;
        }
        while (isalnum(ch));
        temptxt[length] = '\0';
        strcpy(txt, temptxt);
        Search(0, txt, length, name1);
}
```

Fig. 3: Algorithm for Identifier Identification

character whether it is an assignment symbol again or not. This is to differentiate between an assignment symbol and an equal symbol.

```
//symbol "==" or '='
if (ch == '=')
{
 ch = RichEdit1->Lines->Text.c_str()[character++];
 if (ch == '=')  //symbol "=="
     ListBox1->Items->Add(equal1);
 else            //symbol '='
     ListBox1->Items->Add(assignment1);
}
```

Fig. 4: Algorithm for Special Symbol Classification

In sub-C++, usually comments will start with symbol (/*) and end with (*/). The comment function, *comment()* will be called if the lexical analyzer detects the symbol (/*). By using this function, all characters between the comments will be ignored as shown in Fig. 5. The lexical analyzer is capable on detecting a comment in a single line as shown in Fig. 6.

B. The Design of Searching Module

The searching module will start after the lexical analyzer module. It uses the outputs from the lexical analyzer as an input. The purpose of this module is to get the information that will be used in software maintenance such as the information related to global variables, local variables, function and so on. The complete type of information that users can get from this module is shown in Fig. 7.

The searching process is dependent on the grammar or basic syntaxes for sub-C++ language are the same as other programming languages. For example, if users want to get the number of variables, so the syntax for variable will be used as shown in Fig. 8.

In this module also, users can list out and determine each classes whether it is a based or derived class. It also depends on its own syntax. For example, the syntax of based class is

```
Comment(char ch, int &character)
{
   ch = RichEdit1->Lines->Text.c_str()[character++];

   while ((ch != '*')&&(ch != NULL))
   {
      ch = RichEdit1->Lines->
      Text.c_str()[character++];
   }
   if (ch == NULL)
      ListBox1->Items->Add("comment error");
   else
   {
      ch = RichEdit1->Lines->
      Text.c_str()[character++];
      if (ch == NULL)
         ListBox1->Items->Add("comment error");
      else if (ch != '/')
         Comment(ch, character);
   }
}
```

Fig. 5: Algorithm for Comment Identification

```
 if (ch == '/')
 {
   ch = RichEdit1->Lines->Text.c_str()[character++];

   //comment with star
   if (ch == '*')
   {
     Comment(ch, character);
     commentline++;
   }

   //comment one line
   else if (ch == '/')
   {
     //12 for endline |10 for return
     while (ch != 10)
     {
       ch=RichEdit1->Lines->
       Text.c_str()[character++];
     }
     commentline++;
     --character;
   }

   //slash
   else
   {
     ListBox1->Items->Add(div1);
   }
 }
```

Fig. 6: Algorithm for Comment Classification

shown in Fig. 9. The syntax of based class starts with the reserved word *class*, follow by name of class *ClassName*. Then, the body of the class starts with a open curly bracket symbol, and ending with a close curly bracket symbol, and then follow by semi colon symbol. Beside this, the syntax of a derived class is shown in Fig. 10, where a derived type is refer to a public or private class only.

The process of recognizing of these two types of classes is based on the colon symbol(:). For example, if the colon symbol is discovered after the *className*, the class is categorized as a derived class, otherwise it becomes a base class. This module can also detect objects of classes, that is based on their own syntax as shown in Fig. 11.

- List and the number of all variables; global and local including their position
- List of based classes
  - List and the number of based classes
  - List and the number of object for based classes including their position
  - List and the number of derived class
  - List and the number of data members including their position
  - List and the number of function members including their position
- List of derived classes
  - List and the number of derived classes
  - List and the number of object for derived class including their position
  - Type of derived class
  - List and the number of derived class
  - List and the number of data members including their position
  - List and the number of function members including their position
- List of functions
  - List and the number of functions
  - List and the number of local variables
  - List and the number of called functions
- List and the number of global variables including their position

Fig. 7: Searching Module

```
<definition type> :: = <type> <IdList>;
<type>     ::= int | char
<IdList>   ::= <iditem> | <iditem> , <IdList>
<iditem>   ::= <id> | <id> = <value>
<id>       ::= <letter> <restofid>
<restofid> ::=|<validchar>|<restofid><validchar>
<validchar>::= <letter> | <numeral> | _
<letter>   ::= a|b|c|.......|X|Y|Z
<numeral>  ::= 1|2|3|4|5|6|7|8|9|0
```

Fig. 8: Syntax for variable declaration

V. Conclusion

The purpose of this paper is to design a basic information extraction tool for software maintenance. The tool will give users an ability to know the basic information from existing program (sub-C++) such as variables; local and global, a list of base and derived classes, list of functions, objects of

```
class className
{
   classBody
};
```

Fig. 9: Syntax for class

```
class className : deriveType basedClassName
  {
     classBody
};
```

Fig. 10: Syntax for derived class

```
className <objectList>;
```

Fig. 11: Syntax for object class declaration

classes, and so on. Because the tool just produced the basic information, hence it is called the basic software maintenance tool. The tool have two main parts; the lexical analyzer module that can read the input file character by character, and the search module which enables users to obtain basic information from the existing program.

Based on the current ability of this basic tool, we plan to extent the scope of sub-C++ program to become a standard of C++, so that the tool is more compatible with all C++ programs. We can also implement this type of tool to other programming languages like Java and Visual Basic.

Acknowledgment

This research is sponsor by Ministry of Education, Malaysia Government under Research Acculturation Collaborative Grant (RACE).

References

[1] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, 1990.
[2] Phillip A. Laplante. What Every Engineer Should Know About Software Engineering. CRC Press, 2007.
[3] Roger S. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill Higher Education, UK, 2010.
[4] Ian Sommerville. Software Engineering(7th ed.). Addison Wesley, 2004.
[5] Keith H. Bennett and Václav Rajlich. Software maintenance and evolution: a roadmap. In Proceedings of the International Conference on Software Engineering (ICSE'00), pages 73-87, 2000.
[6] IEEE Standard for Software Maintenance. IEEE Std 1219-1993, Jun 1993.
[7] ISO/IEC Standard for Software Maintenance. ISO/IEC Std 14764:1999, 1999.
[8] Václav Rajlich. Software Evolution and Maintenance. In the Proceedings of the Future of Software Engineering (FOSE 2014), pages 133-144, 2014.
[9] Michele Risi and Giuseppe Scanniello. MetricAttitude: A Visualization Tool for the Reverse Engineering of Object Oriented Software. In the Proceedings of the International Working Conference on Advanced Visual Interfaces, pages 449-456, 2012.
[10] Josip Maras and Maja Štula and Ivica Crnkovic. phpModeler - A Web Model Extractor. In the Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pages 660-661, 2009.
[11] Cséri, Tamás and Szügyi, Zalán and Porkoláb, Zoltán. Rule-based Assignment of Comments to AST Nodes in C++ Programs. In the Proceedings of the Fifth Balkan Conference in Informatics, pages 291-294, 2012.
[12] Chen, Haibo and Yu, Jie and Chen, Rong and Zang, Binyu and Yew, Pen-Chung. POLUS: A POwerful Live Updating System. In the Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pages 271-281, 2007.
[13] Sensalire, Mariam and Ogao, Patrick and Telea, Alexandru. Classifying Desirable Features of Software Visualization Tools for Corrective Maintenance. In the Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis 2008), pages 87-90, 2008.