Curtin University

Sri Lanka Institute of Information Technology

Software Metrics (ISAD4002)

Semester 1, 2017

Assignment 2 Report: Class File Parser

Submitted by

Buddhika Jayanarth Benthara Poramba Badalge

Curtin ID: 19201973

SLIIT ID: IT15000286

Table of Contents

Compiling and Running:

Compiling

With all the files from the ClassFileParser folder in the same folder run the following in the terminal:

```
javac *.java
```

Ignore any notes like:

“Note: Some input files use unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.”

Running

With the class file (eg. TestRecursive.class) that needs to be parsed put in the same folder as the compiled program run the following command in the terminal:

```
java ClassFileParser classfilename.class
```

(eg. java ClassFileParser TestRecursive.class)

After that the methods for the user specified class will be displayed. Enter the corresponding number for the method whose tree needs to be viewed. Or enter 9999 to view a non-recursive tree for the full class.

When a method number is entered user will be prompted to enter a number specifies how many levels of recursion the user wants for recursive methods.

Class Tree Format

Eg.

Method: UserRequestedMethod()

Invokes: Method1() ←First Indentation

Method: Method1() ←Second Indentation

Invokes: Method2() ←Third Indentation

Invokes: Method3() ←First Indentation

In this case, the method requested by the user invokes both Method1 and Method3, which is display on the first indentation. If the an invoked method also class another method first it's information is displayed on the second indentation (preceded by "Method: ") and the invocation displayed on the third indentation. Method1 further invokes Method2. Usually the tree ends at the invocation unless it is recursive, in which case it ends with [MethodName Recurring]. It has one design format inconsistency: it will also display method info for empty invoked methods (as in the access flags for invoked methods that don't call other methods).

Functionality

As per the tests done the program meets the following requirements:

- It can differentiate between overloaded methods and displays parameter types so the user can choose the necessary overloaded method to view in the tree.
- It displays callee methods that are invoked multiple times (at the same level of invocation) once.
- It displays a [Recurring] text for methods that that continuously recursively calling either themselves or one another.
- It displays a [Abstract] text for invoked methods that are abstract methods.
- Outputs a call tree that is formatted to use indentation to illustrate invocation level
- Displays the number of unique methods invoked by the user specified method (or for all methods if the full class tree option is selected)
- Missing text displays when a method is not found in the class

On the other hand, a drawback of the program is that it is limited to one class per runtime.

Design

The program was built on the partially implemented ClassFIleParser[1] program provided on Blackboard by Dr. David Cooper (2014). The pre-existing classes were mostly unmodified. The few changes were to CPEntry in order to change the format of Strings the functions there returned. Instruction and Opcode classes were removed.

The classes modified extensively:

ClassFile (from where constant pool parsing ends)

The classes developed are:

Field, Interface, Attribute, Method, InvokedMethods

ClassFIleParser had already parsed classes until the constant pool. Utilising an object of that constant pool, I created an arraylist each for field objects, interface objects, attribute objects, method objects and invoked methods objects.

For the purpose of build the class tree I wasn't too concerned with fields and interfaces (or their attributes). Rather I focused on the rest. My thought process was I would store data read from the datastream and allocate them to my array-lists of objects according to the JVM Speciation for Class File Format[2]. This resulted in an array-list of Method objects (each method object representing a method) which held an array-list of its Attribute objects (specifically its code attributes) which in turn held an array-list its InvokedMethods objects. The InvokedMethod object is where the info for a single method invocation is stored.

In the ClassFile class I developed a method PrintSpecificMethod (and PrintClassTree for the full class tree), which takes in a few inputs and builds the method tree onto the terminal.

Testing

Four classes have been developed for testing purposes:

TestAbstract:

Designed to test if invoked method is abstract, this class has a few abstract methods and a regular method (Method3()) that calls an abstract method.

```
==== Recursive class tree for Method3 ()V ====

Method: ACC_PUBLIC ()V Method3
     Invokes: java/io/PrintStream (Ljava/lang/String;)V println
** Method not found (not in this class): (Ljava/lang/String;)V println
     Invokes: TestAbstract (II)I AbstractMethod1 [ABSTRACT]
          Method: ACC_PUBLIC ACC_ABSTRACT (II)I AbstractMethod1
** Number of Unique Methods and Constructors invoked by method: 2 **
```

TestMissing:

Contains a method that calls a method (method()) from outside the class. Since this program handles only one class it cannot invoke the required method and therefore marks it as missing.

```
==== Recursive class tree for method ()I ====

Method: ACC_PUBLIC ACC_STATIC ()I method
     Invokes: Sum ()V <init>
          Method: ACC_PUBLIC ()V <init>
               Invokes: java/lang/Object ()V <init>
                    Method: ACC_PUBLIC ()V <init>
                         Invokes: java/lang/Object ()V <init>
                              [<init> Recurring]
     Invokes: Sum ()V disp
** Method not found (not in this class): ()V disp
     Invokes: TestMissing ()I method2
          Method: ACC_PUBLIC ACC_STATIC ()I method2
** Number of Unique Methods and Constructors invoked by method: 3 **
```

TestOverloaded:

Designed to see if program can differentiate between overloaded methods. Inside it method() calls 3 different overload method1()'s and displays them separately. Parameters are also shown for each kind.

```
==== Recursive class tree for method ()V ====

Method: ACC_PUBLIC ACC_STATIC ()V method
     Invokes: TestOverloaded (II)V method1
          Method: ACC_PUBLIC ACC_STATIC (II)V method1
     Invokes: TestOverloaded (I)V method1
          Method: ACC_PUBLIC ACC_STATIC (I)V method1
     Invokes: TestOverloaded ()V method1
          Method: ACC_PUBLIC ACC_STATIC ()V method1
** Number of Unique Methods and Constructors invoked by method: 3 **
```

TestRecursive:

This class has a few recursive methods. Some calling themselves and some calling each other recursively. The program will stop recursions at a given user provided recursion level and mark it.

```
Enter how many levels of invocation for recursive methods:
4

==== Recursive class tree for method1 (II)V ====

Method: ACC_PUBLIC ACC_STATIC (II)V method1
     Invokes: TestRecursive (II)V method1
          Method: ACC_PUBLIC ACC_STATIC (II)V method1
               Invokes: TestRecursive (II)V method1
                    Method: ACC_PUBLIC ACC_STATIC (II)V method1
                         Invokes: TestRecursive (II)V method1
                              Method: ACC_PUBLIC ACC_STATIC (II)V method1
                                   Invokes: TestRecursive (II)V method1
                                        [method1 Recurring]
** Number of Unique Methods and Constructors invoked by method: 1 **
```

TestUnique:

This class has al method calling 3 overloaded methods method but in multiple instances at a time. The program ignores the multiples after taking one unique invocation per method and displays. It displays a summary of the number unique methods and constructors in post.

```
==== Recursive class tree for method ()V ====

Method: ACC_PUBLIC ACC_STATIC ()V method
     Invokes: TestUnique (II)V method1
          Method: ACC_PUBLIC ACC_STATIC (II)V method1
               Invokes: TestUnique (II)V method1
                    Method: ACC_PUBLIC ACC_STATIC (II)V method1
                         Invokes: TestUnique (II)V method1
                              [method1 Recurring]
     Invokes: TestUnique (I)V method1
          Method: ACC_PUBLIC ACC_STATIC (I)V method1
               Invokes: TestUnique ()V method1
                    Method: ACC_PUBLIC ACC_STATIC ()V method1
                         Invokes: TestUnique (II)V method1
                              [method1 Recurring]
     Invokes: TestUnique ()V method1
          Method: ACC_PUBLIC ACC_STATIC ()V method1
               Invokes: TestUnique (II)V method1
                    Method: ACC_PUBLIC ACC_STATIC (II)V method1
                         Invokes: TestUnique (II)V method1
                              [method1 Recurring]
** Number of Unique Methods and Constructors invoked by method: 3 **
```

**Both Java and Class files for the test classes are in the Test_Case_Classes folder

Quality

The RSM output can be found in the RSM Output folder. The following is a summary of that report:

~~ Project Quality Profile ~~

Type	Count	Percent	Quality Notice
1	52	17.11	Physical line length > 80 characters
16	1	0.33	Function/class/struct/interface white space < 10.0%
17	10	3.29	Function comment content less than 10.0%
20	1	0.33	File comment content < 10.0%
28	6	1.97	Cyclomatic complexity > 10
31	3	0.99	Class/Struct comments are < 10.0%
38	1	0.33	Exception Handling "try"- "catch" has been identified
46	9	2.96	Function/Class Blank Line content less < 10.0%
47	2	0.66	File Blank Line content < 10.0%
48	1	0.33	Function lLOC <= 0, non-operational function
49	60	19.74	Function appears to have null or blank parameters
50	42	13.82	Variable assignment to a literal number
51	89	29.28	No comment preceding a function block
55	8	2.63	Scope level exceeds the defined limit of 5
121	19	6.25	class name not proper cased
	304	100.00	Total Quality Notices

Number of comments aside I don't feel like there are too many readability issues. There are some functions with a larger scope level but they were essential, for example the class tree printing function. That also lends to the many type 1 notices.

As for the " a function appears to have null or blank parameters" notice, I believe it is due to some variables being initialised during runtime.

References

1) David Cooper (April 30 2014), “ClassFileParser_2014-04-30.zip” Blackboard Curtin University,

https://lms.curtin.edu.au/bbcswebdav/pid-4562073-dt-content-rid-25463678_1/xid-25463678_1 (accessed May 15, 2017)

2) Oracle America, Inc (July 2011), “Virtual Machine Specification ("Specification")

Version: 7: Chapter 4. The class File Format” oracle.com,

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-0-front.html (accessed May 15, 2017).