SOFTWARE METRICS

ISAD4002

ASSIGNMENT 2

Parsing Compiled Code

Fatima Shabbir

19201960

Compiling and Running

If the command-line is being used the the program may be compiled using the following after moving to the source code directory:

```
[user@pc]$ cd Code
[user@pc]$ javac *.java
[user@pc]$ java ClassFileParser
```

The compilation results in a warning due to an exception referenced in the class. This may be ignored as it does not affect the code or the results in any way.
The user will then be prompted to enter the class name and the method name to build to call tree for.

Functionality

Implemented:

- ✓ The code has been implemented to build the call tree for the method that is entered by the user.
- ✓ Both the class and the method for which the call tree is to be constructed shall be entered by the user. If these are invalid, an error message is displayed.
- ✓ Standard input is used to take in both parameters.
- ✓ The methods printed are unique.
- ✓ The total number of the methods called under the method entered are printed.
- ✓ If a method is called recursively it is indicated.
- ✓ If missing classes are initialized and methods from them are called it is marked with [missing].
- ✓ Overloaded methods are printed separately.

Not implemented:

- ✘ Reading from multiple classes
- ✘ Abstract methods

Design

The class CPEntry has been modified to provide variables that can be easily manipulated. The class has not been altered beyond the Strings returned.

The class OpCode was used to reference to OpCodes in the attributes.

The class ClassFileParser has been altered to accept standard input in place of the command line arguments. If the class entered does not contain .class then an error is printed and the program exits. It accepts the class name and passes it on to ClassFile.

All the main methods that are used to parse the class are provided in ClassFile.java. The file includes the methods: makeField(), makeMethod(), makeAttribute, print(), printNested(), getAttrCountIndex(), getMethodIndex(), isMissing() and isRecursive().

ClassFile(), the constructor, acquires bytes before it reaches the fields and then calls the makeField() method in a for loop depending on the field count given by the class file. It then calls makeMethod() method in a for loop constrained by the methods count given by the class file. Once the methods are made it prints out a list of the methods and queries the user for a method to make the call tree for. It calls the print() method to build and print the call tree.

makeField() reads the bytes for the fields and makes up the fields storing them in a String. This method is never used as the fields are not required for the purposes of this assignment.

makeMethod() method is used to put together the method and store them in an array list. In order to complete this it calls a method called makeAttribute that puts together all the attributes inside that method.

makeAttribute() method gets the methods invoked by the methods in the class and stores them in an array list called signatures and forms corresponding array that holds the number of the attributes, called attrCount which is initialized to the size of the method count. This allows the methods to print the attributes from signature by referencing them with attrCount. The attribute is only constructed if it is

Code, else the bytes are read and left alone. If the attribute has another attribute it calls itself recursively.

getInvokedMethods() method gets all the methods invoked by a particular method. A method name is passed to it in order to do this. If the method does not exist in the class provided it prints an error message and exits. GetInvokedMethodsM() is called by this method in order to check those methods call any method. This method then calls itself recursively. The result of both methods is a fully formed call tree for the method entered by the user.

print() method begins printing the call tree. It achieves by printing the method inputed by the user and then passing it to printNested() in order to build the call tree. It acquires the method index in the methods array list via a method called getMethodIndex() and the position in the signatures array list via the getAttrCountIndex() method.

printNested() runs through the signatures array and checks if any method being printed calls methods. If it does it prints them. It first checks if the method is from a missing class via the help of the isMissing() method and then checks if the method is called recursively via the help of the isRecursive() method. In order to build the call tree printNested() calls itself recursively until it reaches the end of the tree.

Testing

Describe your test cases, justify how you designed them, and give the results of your testing. Describe any bugs you are aware of.

Basic test:

The basic test is performed to form the call tree via a class called Basic.class which is included in the zip file under the “TestCases” directory.
It was formed to test the basic functionality of the call tree where a method is called by another method which calls another method. The method called for the expected output is m1();

Code:

```
public class Basic {



public void m1 () {
m2();
}

public void m2 () {
m3();
}

public void m3 () {
}
}
```

Expected output:

```
Basic.m1 ()V
Basic.m2 ()V
Basic.m3 ()V
```

Output achieved:

```
Basic.m1 ()V
Basic.m2 ()V
Basic.m3 ()V
```

Conclusion:

The test was successful.

Recursion test:

The recursion test is performed to form the call tree via a class called Recursion.class which is included in the zip file under the “TestCases” directory.

It was formed to test the recursive functionality of the call tree where a method is called by another method which calls another method which in turn calls the initial method. The method called for the expected output is m1();

Code:

```
public class Recursion {

public void m1 () {
m2();
}

public void m2 () {
m3();
}

public void m3 () {
m1();
}
}
```

Expected output:

```
Recursion.m1 ()V
Recursion.m2 ()V
Recursion.m3 ()V
Recursion.m1 ()V [recursive]
```

Output achieved:

```
Recursion.m1 ()V
Recursion.m2 ()V
Recursion.m3 ()V
Recursion.m1 ()V [recursive]
```

Conclusion:

The test was successful.

Complex test:

The complex test is performed to form the call tree via a class called Complex.class which is included in the zip file under the “TestCases” directory. The test was not performed on real world classes as some of the functionality has bugs such as the inability to handle multiple classes and therefore cannot handle real world classes.

It was formed to test the overall functionality of the parser and its ability to print the call tree. The method called for the expected output is methodB();

Code:

```
public class Complex {

public void methodA() {
methodB();
methodB();
}

public void methodA(int s) {
methodA();
}

public void methodB() {
methodA();
methodA();

for(int i = 0; i < 10; i++)
methodC();

m2(2);

methodA(5);
}

public void methodC() {
m3();
m3();
}

public void m2 (int s) {
methodB();
Test t1 = new Test();
t1.m3();
m2(2);
}

public void m3 () {}
}
```

Expected output:

```
Complex.methodA ()V
Complex.methodB ()V
Complex.methodA ()V [recursive]
Complex.methodC ()V
Complex.m3 ()V
Complex.m2 (I)V
Complex.methodB ()V [recursive]
Test.<init> ()V [missing]
Test.m3 ()V [missing]
Complex.m2 (I)V [recursive]
Complex.methodA (I)V
PrintStream.println (Ljava/lang/String;)V [missing]
Complex.methodA ()V [recursive]
```

Output achieved:

```
Complex.methodA ()V
Complex.methodB ()V
Complex.methodA ()V [recursive]
Complex.methodC ()V
Complex.m3 ()V
Complex.m2 (I)V
Complex.methodB ()V [recursive]
Test.<init> ()V [missing]
Test.m3 ()V [missing]
Complex.m2 (I)V [recursive]
Complex.methodA (I)V
PrintStream.println (Ljava/lang/String;)V [missing]
Complex.methodA ()V [recursive]
```

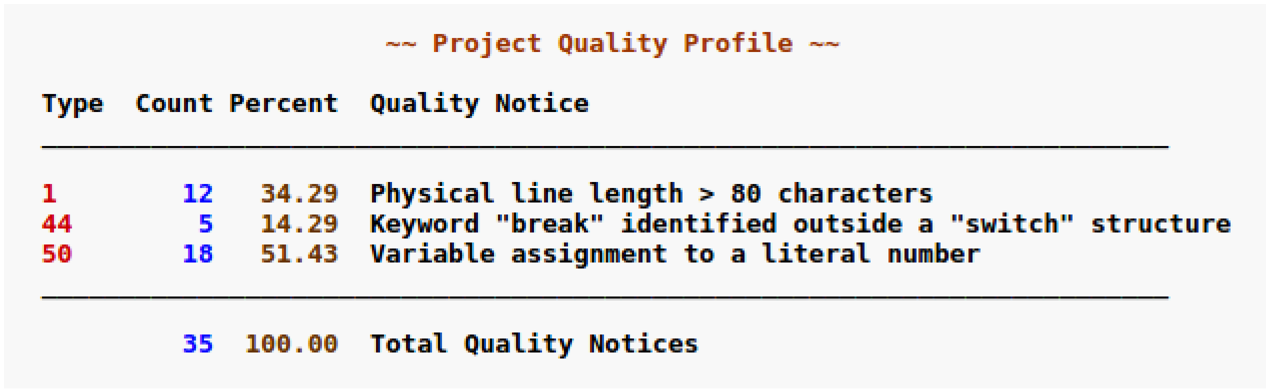
Conclusion:

The test was successful.

Quality

The files test for quality are ClassFile and ClassFileParser and those are the only two files that have experienced any major changes from what Dr. David Cooper has provided us with. Both files are provided in the folder called Quality in the zip file.

Below is the code quality profile for **Class File:**



```
                    ~~ Project Quality Profile ~~

Type   Count Percent  Quality Notice
________________________________________________________________________

1         12   34.29  Physical line length > 80 characters
44         5   14.29  Keyword "break" identified outside a "switch" structure
50        18   51.43  Variable assignment to a literal number
________________________________________________________________________

          35  100.00  Total Quality Notices
```

As illustrated by the report there are three notices generated summing up to a total of 35 notices.

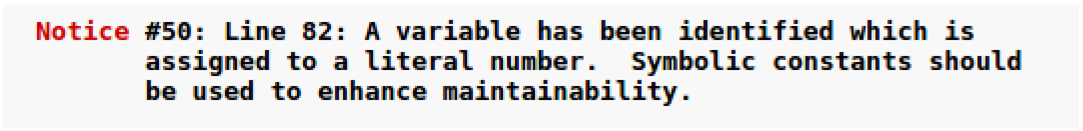
Reasoning and Explanation

Notice #1:

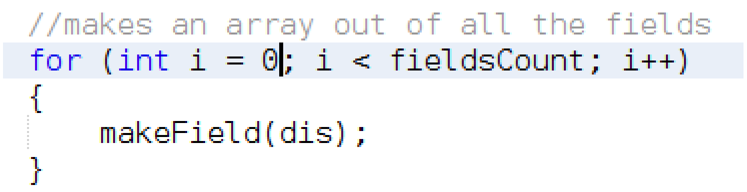
These notices are generated for print statements or method signature where the methods throw exceptions thus adding to the length of the line. Since both these situations are unavoidable the notice cannot be dealt with.

Notice #50:

These notices are generated for literal numbers assigned to variables. And example of the code and the corresponding notice is given below.



```
Notice #50: Line 82: A variable has been identified which is
       assigned to a literal number.  Symbolic constants should
       be used to enhance maintainability.
```



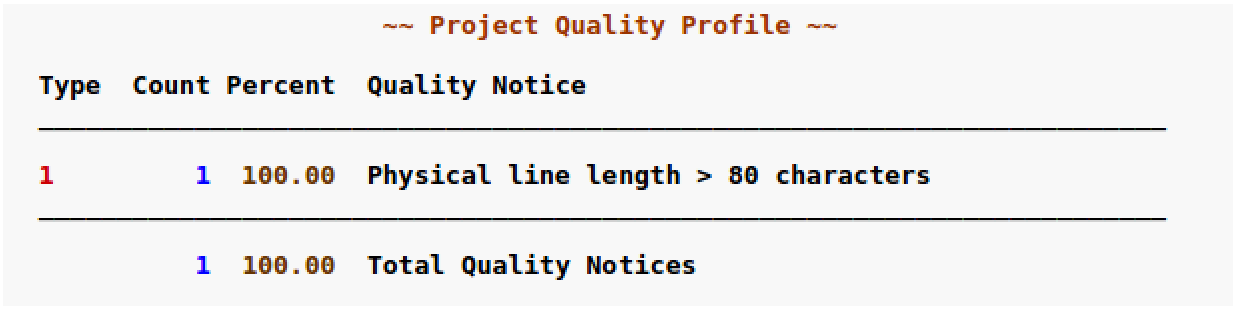
```
//makes an array out of all the fields
for (int i = 0; i < fieldsCount; i++)
{
    makeField(dis);
}
```

As it can be observed the zero assigned to i in the for loop is generating the notice. Since this does not seem like a major quality issue that hinders the working or the maintainability of the code it is not dealt with.

Notice #44:

The break keyword is essential to the logical of the call tree printing and therefore the quality notice has been not dealt with.

Below is the code quality profile for **Class File Parser:**



```
                    ~~ Project Quality Profile ~~

Type  Count Percent  Quality Notice
________________________________________________________________________

1         1  100.00  Physical line length > 80 characters
________________________________________________________________________

          1  100.00  Total Quality Notices
```

As illustrated by the report there is only one notice generated.

Reasoning and Explanation

Notice #1:

The notice is generated for the method signature where the method throws exceptions thus adding to the length of the line. Since the situation is unavoidable the notice cannot be dealt with.

Credits

All code except for the one written by Dr. David Cooper for the ClassFileParser provided on OASIS has been written solely by me.
Credits for all code written by Dr. David Cooper belong to him.