Extracting and comparing variables on Java source code

Mirko Raimondo Aiello, Andrea Caruso
Department of Mathematics and Informatics
University of Catania
Catania, Italy
Email: mremaiello@gmail.com, andrea299024@gmail.com

***Abstract*—We have realised a Java source code analyser that firstly extracts code from a repository and then analyses the code to gain some knowledge. In particular, for each method of each class the internal variables used are automatically determined. For Java classes, we find method bodies and variable declarations by using regular expressions. The tool has been developed by means of Java and MrJob. Python with the MapReduce model have been used to perform data analysis in a distributed manner.**

***Index Terms*—Parallelization, Java, Python, MapReduce, algorithms, data dependence**

I. Introduction

We have developed a system that analyses different versions of a software system, which is automatically extracted from a git hub repository [1], [11]. Essentially, given two versions of a Java class, we gather: the name of the classes, method names within classes, the number of starting and ending lines of methods and variables used by the methods.

The analysis is performed by two different tools. The first tool, developed in Java, takes as input the source code and get classes, methods and variables. The second tool, developed in Python, analyses the output of the first tool and gets the variables within each method. To mine the data from a set of Java classes, we use regular expressions, a sequence of characters that define a search pattern, mainly used in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

Since the said computation can become expensive with the increasing quantity of code to analyse, the second tool can be executed in a distributed manner using Hadoop, an Apache framework providing the MapReduce model, for providing support to distributed systems accessing big data [8], [14], [18]. Of course distribution depends on the underlying infrastructure and configuration, for which other support may be needed [3], [4], [12], [13]. To use all the advantages of the Python language, we adopt the MrJob toolkit that helps to develop Hadoop programs and test them locally [2].

Finally, the data obtained from the computation are saved to a .csv file into the output directory, and can be read using tools like CSV Viewer. Both the first and second tool will be subject to evaluation tests, to calculate execution time and obtain some results from a chosen Java repository.

The proposed analysis can be valuable when examining existing code in view of refactoring opportunities [9], [10], [16]. Moreover, it is often necessary for developers to find essential characteristics and metrics which give a view on the internal characteristics, such as design patterns used [7], [15], or performance related and dependence issues [6], [17]. This is even valuable for the problem of validating software systems [5].

The following of this paper firstly describes the problem at hand, then describes the proposed solution in detail, then results are shown, and finally conclusions are drawn.

II. Requirements analysis

Requirements analysis in systems engineering and software engineering encompasses tasks that determine the needs or conditions to meet for the product. As said previously, the entire software product is divided in two tools, for each the following requirements have been identified.

- ***Analysis of Java source code***: the first tool will be able to analyse Java source codes, obtaining the desired data, i.e. the names of classes, methods within the classes, the number of start and end line of each method, and the variables used by such methods. The results are stored in appropriate data structures.
- ***Reading and writing information into files***: the second tool will be able to read .txt files placed in a specific directory. These files contain the bodies of methods previously extracted. The extracted data will be placed in .csv and .txt files. In particular, the overall structure of the examined Java files will be placed in a .csv file, while the bodies of the identified methods will be included in a .txt file.
- ***Extract variables***: the second tool will be able to extract the names and types of the variables used by each method, by using the MapReduce paradigm. The variables will be associated with the method that uses them.
- ***Comparison of files***: the second tool should analyze the variables that are used within methods, with the purpose of tracking changes between two different versions of the same method. The difference will be performed in both directions, i.e. by identifying variables that can be within the first version of the method and not in the other version and vice versa.

III. Proposed Approach

The development of the proposed tools involved different programming languages combined together in a pipeline with a bash script. For the Tool 1, used for the pre-processing, we used Java. For the Tool 2, that computes the differences, we used Python with MrJob toolkit. Figure 1 shows a general overview of the developed components.

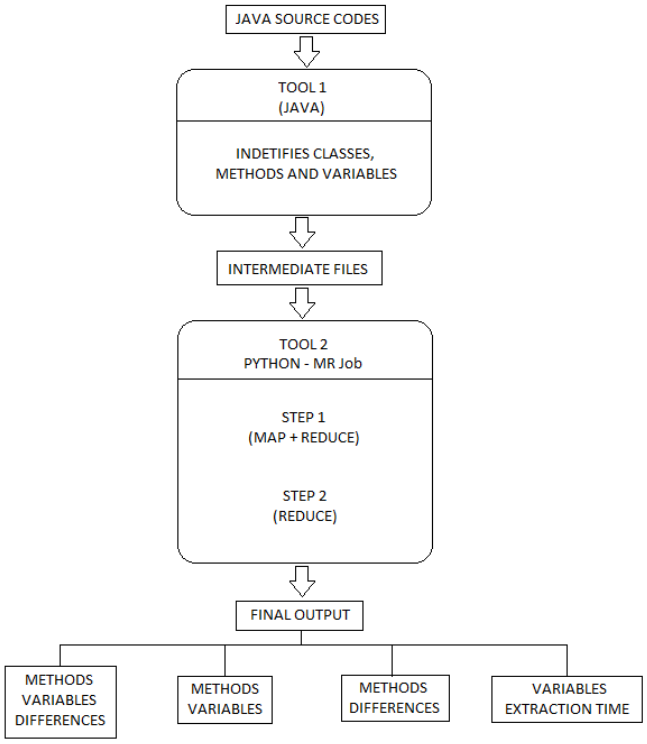

Fig. 1. Java Source Code Analyser workflow.

All the algorithms of the said tools described here are valid for an arbitrary number of Java classes. However, to make it easier to understand the algorithms used in this project, all the examples and the related images refer to the analysis of the Java files in Figure 2 and Figure 3.

```
public class Fact
{
    public void Fact() {
        final int NUM_FACTS = 100;
        for(int i = 0; i < NUM_FACTS; i++)
            System.out.println( i + "! is " + fact_rec(i));
    }

    public int fact_rec(int n) {
        int i = x - 1;

        while ( i > 0 ) {
            x = x * i;
            i --;
        }

        return x;
    }
}
```

Fig. 2. Recursive Factorial Java Code used as an example.

```
public class Fact
{
    public void Fact()
    {
        final int NUM_FACTS = 100;
        for(int i = 0; i < NUM_FACTS; i++)
            System.out.println( i + "! is " + it_fact(i));
    }

    public int it_fact(int n)
    {
        int result = 1;
        for(int i = 2; i <= n; i++)
            result *= i;
        return result;
    }
}
```

Fig. 3. Iterative Factorial Java Code used as an example.

IV. The first Tool

The analysis of the class code has been performed by a ParseText class taking as parameters: (i) an input string holding the code of the class under analysis, and (ii) a string that represents the output folder.

The source code of the class under analysis is read line by line, and regular expressions (regex) are used on each line to identify the signature of the methods, and the respective row in which they occur. Since the lines of code holding method signatures can have different forms (e.g. the constructor does not have a return type), then there will be more regex, contained in a list (classRegexList). By knowing all the starting lines of code for methods, it is possible to extract the ending lines of the methods by simply calculating the differences between the start line of the i-th method and the beginning of the next method.

```
for (Pattern regex : classRegexList)
{
    matcher = regex.matcher(line);
    if (matcher.find())
    {
       if (!matches.contains(i))
            matches.add(i);
    }
}
```

Fig. 4. Fragment of code used for the extraction of information.

Once these two pieces of information have been obtained, we can analyse the method body. Obviously, the code will check that the structure of the method is compliant with the Java specifications. The next step uses regular expressions again, and by manipulating strings (by means of replace, replaceAll, trim) extracts the class name, the signature and the method body.

The output generated by the first tool consists of a number of files in two categories.

- ***A group of txt files***, one for each method found on the analysed code, containing the name of the reference

```
((public|private|protected|static|final|native|synchronized|
  abstract|transient)+\s)+[\$_\w]+\([^\)]*\)

((public|private|protected|static|final|native|synchronized|
  abstract|transient)+\s)+[\$_\w\<\>\w\s\[\]]*\s+[\$_\w]+\\([^\)]*\))
```

Fig. 5. Regex used to extract, respectively, the constructor and the methods of a class.

```
((byte|short|int|long|float|double|boolean|char|
  String)+\s)+([a-zA-Z_$][\w$]*)(\[\])*
```

Fig. 6. Regex used to extract variables.

class, the start and the end line of the method within the class, and the method body. A sample output is shown in Figure 7.

- ***A cmList.csv file*** containing the list of the identified methods, variables, starting and ending lines as shown in Figure 8.

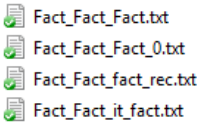

Fig. 7. Example of output files for the Tool 1.

```
className, startLine, endLine, signature
Fact-3-7-public void Fact()
Fact-9-18-public int fact_rec(int n)
Fact-3-8-public void Fact()
Fact-10-16-public int it_fact(int n)
```

Fig. 8. Example of generated cmList.csv file

V. The second Tool

The initial step of the second Tool is to scan the directory in which the first Tool created the .txt files containing the methods bodies. When Tool 2 identifies the files, it stores the filenames in a list called "names". Furthermore, the number of files taken as input is stored in the variable "n_methods".

```
for filename in glob.glob('*.txt'):
    sys.stdin = open(filename, 'r')
    filename=filename.split(".")[0]
    names.append(filename)
    n_metodi=n_metodi+1
    sys.stdin.close()
```

Fig. 9. Fragment of code used for reading file names.

Afterwards, the previously listed files are opened. The ClassFind job is launched on each of them using the run() command.

The MapReduce job consists of two main steps. The first step consists of a map function and a reduce function, with the goal to detect variables and insert them into pairs "methodName, [variablesList]". The second step consists in a further reduce function that has as the goal to calculate variables differences between the examined methods.

```
for filename in glob.glob('*.txt'):
    sys.stdin = open(filename, 'r')
    ClassFind.run()
    sys.stdin.close()
```

Fig. 10. Fragment of code used for parsing files.

```
class ClassFind(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper,
                   reducer=self.reducer),
            MRStep(reducer=self.reducer_difference)
        ]
```

Fig. 11. Fragment of code for the first step.

The mapper detects the presence of variables by means of regular expressions. Each regular expression identifies a particular sequence of characters to search for in the text.

```
def mapper(self, _, line):
    global var_elapsed
    var_start_time = timeit.default_timer()
    filename=os.environ['map_input_file']
    filename=filename.split(".")[0]
    for word in SIMPLE_VARIABLES_RE.findall(line):
        word = f7(word)
        word = ' '.join(word[1:])
        yield filename, word
    for word in COMPLEX_VARIABLES_RE.findall(line):
        yield filename, word
    var_elapsed=var_elapsed+(timeit.default_timer() - var_start_time)
```

Fig. 12. Fragment of code of the mapper.

The first reducer receives pairs "methodName, variableName" detected by the mapper and groups them according to the "methodName" key, forming "methodName, [variablesList]" pairs.

```
def reducer(self, key, values):
    var_start_time = timeit.default_timer()
    yield key, list(values)
```

Fig. 13. Fragment of code of the first reducer.

The second reducer is more complex than the previous. It takes as input the identification key of the current method and the list of variables associated with it. Variables are read by using the "next" command and stored in the "variables" string, then used for printing the "variables.txt" file, which will contain pairs "methodName, [variablesList]". The dict type (key-value dictionary) structure "variables_list" is used to store "methodName, [variablesList]" pairs. The variable "count" is a counter incremented whenever the MapReduce

process has been executed on a method. When it reaches the value “n_metodi”, then all methods have been processed, and the used variables have been identified and stored in “variables_list”. Now differences can be easily computed.

Leveraging the “itertools” library, it is possible to calculate all combinations of pairs of methods, so as to compare each method with all the others, which is the task of the “combinations” function.

```
def reducer_difference(self, key, values):
    global variables_list,names,count,variables,differences,output
    b=next(values)
    b=toUnicode(b)
    variables=variables+"METHOD: "+names[count]
                       +"\nVARIABLES: "+str(b)+"\n\n"

    variables_list.setdefault(names[count],b)
    count=count+1

    if(count==n_metodi):
        for pair in itertools.combinations(names,r=2):
            output=output+"METHOD: "+pair[0]+"\n"
                         +str(variables_list[pair[0]])+"\n\n"
            output=output+"METHOD: "+pair[1]+"\n"
                         +str(variables_list[pair[1]])+"\n\n"
```

Fig. 14. Fragment of code of the second reducer.

The calculation of the differences is managed by Counter type structures, which allow to take into account the presence of variables with the same name. Let us suppose, for example, that a method has 2 i named variables, and the second method has only one, the difference between the first and the second method will result in one i variable.

```
c1=Counter(variables_list[pair[0]])
c2=Counter(variables_list[pair[1]])
diff1=c1-c2
diffList1=list(diff1.elements())

diff2=c2-c1
diffList2=list(diff2.elements())

output=output+"DIFFERENCE: "+pair[0]+" - "
            +pair[1]+"\n"+str(diffList1)+"\n\n"
output=output+"DIFFERENCE: "+pair[1]
            +" - "+pair[0]+"\n"+str(diffList2)+"\n\n\n"

differences=differences + "DIFFERENCE: "+pair[0]
                        + " - "+pair[1]+"\n"
differences=differences+str(diffList1)+"\n\n"
differences=differences + "DIFFERENCE: "+pair[1]
                        + " - "+pair[0]+"\n"
differences=differences+str(diffList2)+"\n\n\n"
```

Fig. 15. Fragment of code used for calculation of the differences.

The difference is performed in the same way also in the opposite direction. Then, for each pair of methods, two differences are generated. Different prints for various output files are managed along the entire process. Finally, the various strings that contain the results are written on ad-hoc files.

The variables-finding function of the Tool 2 has been implemented as a variant of Tool 1, hence both tools detect methods variables and associate them to the respective methods. Nevertheless, the two tools achieve this goal in different ways. Tool 1 uses a classical sequential approach without employing the MapReduce paradigm, whereas Tool 2 uses Python and MrJob as described in this section.

The Tool 2 generates four output files.

- ***output.txt***: created by redirecting the standard output, it is a file containing, for each pair of analysed methods, the name of the method, the set of variables of each method and the two differences between the sets of variables. The output is shown in Figure 16.
- ***variables.txt***: contains the list of the identified methods and their variables, as shown in Figure 17.
- ***differences.txt***: shows the differences between the sets of variables only, as shown in Figure 18.
- ***time.txt***: contains a string giving the time in seconds spent to search all the methods variables, as shown in Figure 19.

```
METHOD: Fact
['int i', 'int NUM_FACTS']

METHOD: fact_rec
['int i']

DIFFERENCE: Fact - fact_rec
['int NUM_FACTS']

DIFFERENCE: fact_rec - Fact
[]
```

Fig. 16. Example of output.txt file of the Tool 2.

```
METHOD: Fact
VARIABLES: ['int i', 'int NUM_FACTS']

METHOD: fact_rec
VARIABLES: ['int i']
```

Fig. 17. Example of variables.txt file of the Tool 2.

```
DIFFERENCE: Fact - fact_rec
['int NUM_FACTS']

DIFFERENCE: fact_rec - Fact
[]
```

Fig. 18. Example of differences.txt file of the Tool 2.

```
TIME ELAPSED (VARIABLES EXTRACTION AND INSERTION IN LISTS): 0.00182867050171
```

Fig. 19. Example of time.txt file of the Tool 2.

VI. Results and Timing Analysis

The configuration used for the test is the following.

- LAPTOP: ASUS A56C
- CPU: Intel (R) Core (TM) i7-3537U @ 2.00GHz
- RAM: 4.00 GB
- OS type: 64-bit
- OS: Linux Ubuntu 16.04 LTS

The two tools show similar results from a reliability standpoint. Both seek the variables by means of regular expressions using the same approach. All internal variables of methods of the code under analysis are detected successfully.

Extraction times for finding the internal variables of the methods (and inclusion in key-value lists, in the case of Tool 2) were compared. Execution times are listed in Table I.

Number of files	Tool 1 (Java)	Tool 2 (Python)
2	0.004s	0.001s
3	0.006s	0.002s
4	0.007s	0.002s
5	0.009s	0.003s
6	0.011s	0.004s
6	0.012s	0.005s
8	0.013s	0.006s
9	0.013s	0.007s
10	0.014s	0.009s

TABLE I
ANALYSED EXECUTION TIME STATS.

Both tools show linear progression for the measured timings, increasing with the number of methods for which variables are extracted. In general, the Tool 2 is quicker than the Tool 1. The performance results are summarised in the graph of Figure 20 (by using gnuplot [19]).

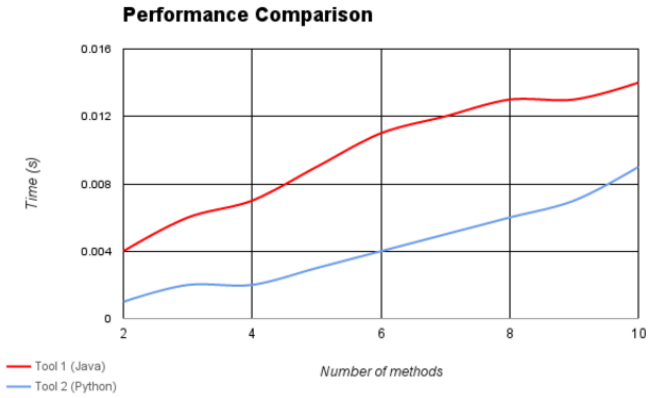

Fig. 20. Graph with the execution times in relation to the total number of methods analysed.

VII. CONCLUSIONS

This paper described a pair of tools that analyses Java source code and extract from them some relevant statistics, such as method's names, signatures, starting and ending lines and variables within methods. The tools have been developed in Java and Python and a comparison of the results in terms of execution time has been shown.

REFERENCES

[1] Github. https://github.com.

[2] Mrjob v0.5.3. http://pythonhosted.org/mrjob.

[3] F. Bannò, D. Marletta, G. Pappalardo, and E. Tramontana. Tackling consistency issues for runtime updating distributed systems. In *Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, Atlanta, GA, April 2010.

[4] G. Borowik, M. Wozniak, A. Fornaia, R. Giunta, C. Napoli, G. Pappalardo, and E. Tramontana. A software architecture assisting workflow executions on cloud resources. *International Journal of Electronics and Telecommunications*, 61:17–23, 2015.

[5] A. Calvagna and E. Tramontana. Automated conformance testing of Java virtual machines. In *Proceedings of Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE, July 2013.

[6] A. Calvagna and E. Tramontana. Delivering dependable reusable components by expressing and enforcing design decisions. In *Proceedings Of Compsac*, pages 493–498, Kyoto, Japan, 2013. IEEE.

[7] S. Cicciarella, C. Napoli, and E. Tramontana. Searching design patterns fast by using tree traversals. *International Journal of Electronics and Telecommunications*, 61(4):321–326, 2015.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

[10] R. Giunta, G. Pappalardo, and E. Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings Of ACM Symposium on Applied Computing (SAC)*, pages 1866–1868, Riva Del Garda (Trento), Italy, March 26-30 2012.

[11] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[12] C. Napoli, F. Bonanno, and G. Capizzi. Exploiting solar wind time series correlation with magnetospheric response by using an hybrid neuro-wavelet approach. *Proceedings of the International Astronomical Union*, 6(S274):156–158, 2010.

[13] C. Napoli, F. Bonanno, and G. Capizzi. An hybrid neuro-wavelet approach for long-term prediction of solar wind. *Proceedings of the International Astronomical Union*, 6(S274):153–155, 2010.

[14] C. Napoli, E. Tramontana, and G. Verga. Extracting location names from unstructured italian texts using grammar rules and mapreduce. In *Proceedings Of the International Conference on Information and Software Technologies (ICIST)*, volume 639, pages 593–601, Druskininkai, Lithuania, October 13-15 2016. Springer.

[15] G. Pappalardo and E. Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *Proceedings Of ACM Symposium On Applied Computing (SAC)*, pages 1591–1596, Dijon, Francia, 23-27 Aprile 2006. ACM.

[16] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, 2006.

[17] H. Washizaki and Y. Fukazawa. Dynamic hierarchical undo facility in a fine-grained component environment. In *Proceedings of International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 191–199. Australian Computer Society, Inc., 2002.

[18] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[19] T. Williams and L. Hecking. Gnuplot, 2003.