GitHubAnalyser: a Tool Detecting Class Correlations on Git Repositories

Gaetano Cammariere, Massimiliano Portelli, Placido Russo
Department of Mathematics and Informatics
University of Catania
Catania, Italy
Email: gaetano.cammariere@outlook.it, massimiliano.portelli@gmail.com, russo.placido@gmail.com

***Abstract*—We have realised a tool, dubbed GitHubAnalyser, performing data mining and analysis of GitHub repositories in order to gain several statistics on Java classes. The sought statistics aim at highlighting the correlation between classes, detected from the simultaneous occurrence of changes on a repository. The tool has been developed using MetricMiner2, a Java mining library, and MrJob that uses Python with the MapReduce model to compute data analysis in a distributed and parallel manner.**

I. Introduction

The proposed GitHubAnalyser tool aims at helping developers to extract three statistics from a Java code repository: (i) the strongly related classes that happen to be modified at the same time, (ii) how many times most of the classes (percentage) were modified together, and (iii) for each class how often has been modified with any other class. Such metrics provide developers with a representation of the software system and can point them to further analysis aiming at improving the modularity of the system, e.g. by means of refactoring metrics and tools [1]–[9].

GitHub is a hosting service for source code based on Git, a version control system for software projects. It simplifies the code sharing and collaboration among projects. The fundamental unit of a repository is the commit, a set of related changes in a repository from which it is possible to derive representation of code state at a given moment in the time. To mine data from a repository we use the Java framework MetricMiner 2 [10] that helps developers with the mining of software repositories. Using this framework we are able to extract some information about commit like date and time of push, author, modified files and the differences among the states of each file.

Since the computation of the statistics becomes expensive with the increasing of the quantity of code to analyse, computation can be executed in a distributed manner using Hadoop, an Apache framework inspired to MapReduce for the support to distributed applications with high access to data [11], [12]. To use all the advantages of the Python language, we use the MrJob toolkit that helps to develop Hadoop programs and test them locally.

Finally, with GnuPlot [13], the data obtained from the computation are visualised in human readable graphs.

II. Gathered Statistics

Through the analysis of a repository, the proposed tool is able to produce as output three statistics which help the developers with their job. The parameters needed for each statistic can be configured in a specific setting file, "settings.ini", which contains all the necessary parameters for the execution of the tool. The full list of parameters is specified in the section II-A. The three statistics are explained below.

The first statistic produces as output a file, "output1.tsv", having the list of the modified classes during a given time range, together with the information of date and time of their commit. The temporal range is set through two parameters which correspond to the two temporal instants that are the limits of the range. The parameters in the file "settings.ini" are *first_statistic_time_inf* and *first_statistic_time_inf*, in the format $dd/mm/yyyy - hh : mm$.

The second statistic produces as output a file, "output2.tsv", having the percentage of modified classes for each commit, given as a percentage of minimum threshold. The modified classes are the classes that have been modified in the same commit, at the same time, and the percentage is relative to the total of classes present in that temporal instant in the repository. The threshold is used to show only the commit whose percentage is above the threshold. The threshold parameter need to set in the file "settings.ini" as parameter *second_statistic_perc*.

The third statistic produces as output file "output3.tsv", having the classes matrix with their frequency of changes. I.e. for each class in the repository, it shows the number of times that it has been modified together with another class. This statistic has no input setting parameter, however two parameters are later used for the visualisation of relative graphs, because the repository could be very large, with many classes.

A. Useful Settings

The tool configuration is given according to the file "settings.ini" that allows the setting of input parameters in the form "key=value". The list of parameters is as follow.

1) ***repository_path***: the location, a local folder or a http/https address, of the repository;
2) ***branch_name***: the branch name of the repository to analyse. The default value is "master" and is also pos-

sible to choose more than one branch name to analyse by separating names by a comma;
3) ***first_statistic_time_inf***: the lower limit used in the first statistic. The format is $dd/mm/yyyy - hh:mm$;
4) ***first_statistic_time_sup***: the upper limit used in the first statistic. The format is $dd/mm/yyyy - hh:mm$;
5) ***second_statistic_perc***: the percentage threshold used in the second statistic. The default value is "0" to show all the modified classes in the commits;
6) ***third_statistic_n_classes***: the number n of graphs to display for the third statistic. Accordingly only the most expressive n classes will be shown in the graph. The default n value is "10";
7) ***third_statistic_threshold***: the modifications threshold used to display the graphs for the third statistic. It allows us to discard the classes whose number of modifications is under the threshold. The default value is "5".

III. Basic Concepts for Processing Repositories

The development of the proposed tool involved several programming languages combined together in a pipeline with a bash script. Figure 1 shows the essential flow of execution. For the mining of repository we use a Java program, based on the Java MetricMiner library, whereas for computing the statistics we use Python, with MrJob toolkit and a Gnuplot script for the visualisation of the results.

A. Mining and pre-processing in Java

The first phase consists of the download of the Java repository and the analysis of the metadata provided by GitHub. Because MetricMiner2 needs a local copy of the repository, before the preprocessing we need to clone the online repository. This is obtained using the "JGit" library that clones the source code using the Git API.

Using the GitHub's metadata we can find the line and the name of the file that contains a modification, and then all the modified classes in that file. MetricMiner2 analyses the repository's metadata subdividing them by commit. For each commit we obtain the timestamp and a list of "Modification". Every "Modification" represents a single file of the project with the updated source code and other information like the added rows and the removed rows. MetricMiner2 builds a tree where a single node represents a Java class with the relative methods. The Java parser inside MetricMiner2 gives the line number of the beginning of a class and we also calculate the line number of its end. With these limits for each class and the line numbers of modified rows derived from a "Modification", we can find the classes that contain at least one modification.

With a single commit, we can obtain only the information about the modifications compared to the previous commit. On account of this, we need to create two maps key/value, respectively, the analysed actual commit and the previous commit. With the two maps we can get all the information about the state of the repository during the project lifecycle. The map contains the information related to the commit with all the Java file in the repository, where the key is the path of

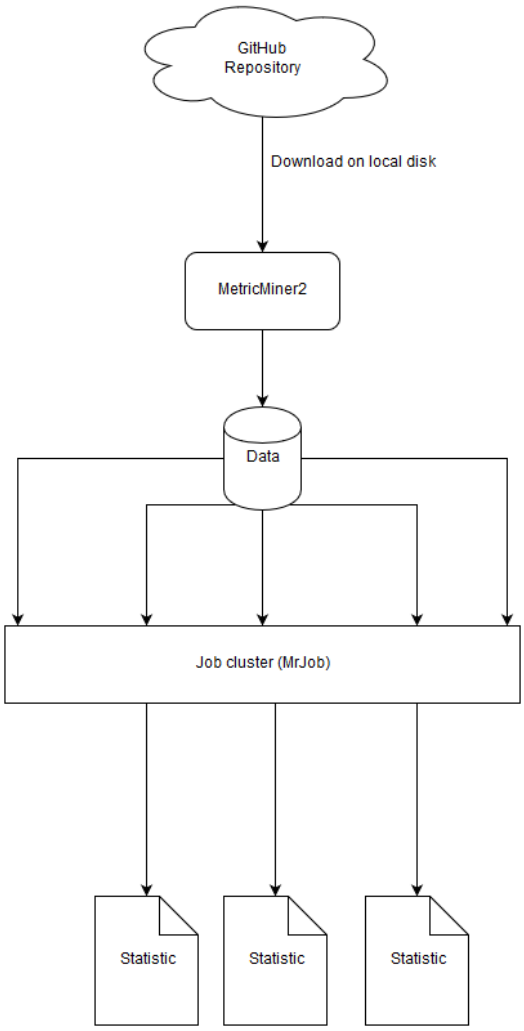

Fig. 1. GitHubAnalyser workflow

the file and the value is the list of classes in the file with a 0 or 1 if the class was modified in the commit or not. The analysis starts from the first commit in the repository in time order with the structure that will be filled with the list of modified classes.

The final result of the preprocessing will be the entire history of the repository's lifecycle subdivided by commit where for every commit we will have its timestamp, all the classes for the timestamp and if some classes are modified a 0 or 1 for each one. These data will be aggregated in a file called "commitsLog.tsv" and they will be processed in the next phase.

The pre-processing produces another output files called "classesList.txt" that contains the list of all class presents in the repository from the initial creation to the last commit. This file is represented in Figure 2.

During the pre-processing some borderline cases are encountered and addressed, as follows.

- If the commits have more than 200 modified files, MetricMiner2 cannot process it and then throws an exception

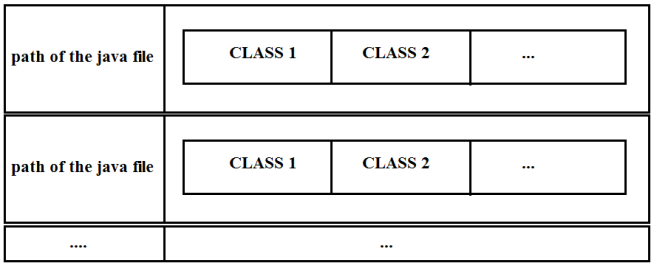

Fig. 2. Graphic representation of a map

and discards the file from the analysis.

- If a single difference file for a single Java file is longer than a threshold (set by MetricMiner2 as 10000 characters) the file content is replaced by a string "TOO BIG" and then the modification cannot be analysed. For this case we choose to consider all the classes in the file modified.

GitHub also presents some limitation in difference files (diff):

- a diff file cannot have more than 1500 lines or more than 100 KB of raw data
- the maximum number of files in a single diff is 300
- the total size of diff file in all files of a view cannot exceed 10000 lines or 1 MB.

By means of the said Java based pre-processing we can obtain data from the repository that we will use in the Python distributed processing. The data obtained are aggregated in two files:

- ***commitsLog.tsv*** that contains the history of the repository subdivided by commit with the relative timestamp, the relative classes and for each class a zero or a one if it has been modified.
- ***classesList.txt*** that contains the list of all classes present in the repository starting from its creation.

B. Computing Statistics

After the data have been extracted from a GitHub repository, they are processed in order to have statistics about the changes made to the source code, specifically on changes to Java classes.

Three Python scripts obtain three statistics. The structure of each script is represented by the implementation of a class derived from MrJob, with two methods inside: *mapper* and *reducer*. These methods are essential in order to define the behaviour according to the paradigm of MapReduce.

Below is the implementation logic of the two methods described for each statistic:

1) ***mapper***: it creates a map that has as a *key* the "timestamp" of a commit and as *value* the "name" of a class that has been modified in the commit;
   reducer: it returns the pairs "date" of the commit, "list" of classes changed in the commit;
2) ***mapper***: it creates a map that has as key the "timestamp" of a commit and as value the "name" of a class present in the source code at the time of the commit, along with a "binary value" indicating whether the class changed in the commit;
   reducer: it returns the pairs "date" of the commit, "percentage" of classes modified in the commit, along with the "list" of changed classes;
3) ***mapper***: given a reference class (which will be one of the class of the repository), it gives as key the "name" of a class modified in the same commit as the reference class and as value "1";
   reducer: it returns the pairs "name" of a class, "number" of times that the class has been modified along with that of reference.

The three scripts produce three output files, called "output1.tsv", "output2.tsv" and "output3.tsv", calculated for the three statistics.

Since the third statistic produces output in a file for each class containing the rate of change of classes compared to the analysed class, another Python script generates a square matrix having rows and columns for the names of all classes present in the repository. Each cell (i, j) indicates the number of times that the $i-th$ class has been changed simultaneously with the $j-th$ class.

This matrix is further processed by another Python script to create two other output files ("filtered_classes_matrix.tsv" and "most_modified_list.txt"), which respectively contain the array of classes sorted in accordance with a minimum frequency threshold and the list of classes modified the most.

The statistical processing is structured to take place in a parallel manner and distributed by using MrJob. In fact the whole calculation was made to run on multiple machines, after proper configuration of a Hadoop cluster.

C. Results visualisation

The result of the calculation of the three statistics is a set of output files in *tsv* format, tab-separated value, that summarise the results of the tool. Furthermore, some graphs are produced to facilitate the user understand the results. This is the list of files produced:

- ***output1.tsv***: it refers to the first statistic and shows the time and date for each commit in the temporal range selected and the list of modified classes during this commits.
- ***output2.tsv***: it refers to the second statistic and shows the commits that the percentage of modified classes is over the input threshold. For each of this commits, identified by time and date, it shows the percentage of modified classes and the list of this classes.
- ***output3.tsv***: it refers to the third statistics and shows the table of all the classes of the repository. Each position (x, y) represent how many times the classes x and y have been modified together.
- ***filtered_classes_matrix.tsv***: it is a table of filtered classes that shows only the significant values of the output3.tsv file. The significance is given by the values that are over the input threshold set in the settings file.

- ***most_modified_list.txt***: it is the list of n most modified classes in the repository, with n value sets in the settings file.

Finally, the execution of the *graphs.sh* script produces this three sets of graphs:

- A graph that shows, only for the commits that exceeds the percentage of modification set in the settings file, the percentage of modified classes compared to the total of classes present in that commit in that temporal instant.
- A set of n graphs, n set in the settings file, that show the frequencies of modifications associated to the first n classes for number of total modification.
- A heat map that shows the number of concurrent modifications for classes, filtered by a specific value in the settings file.

Figure 3 shows the results of the analysis of the *wavefrontHQ/java* repository [14], a repository chosen for testing phase.

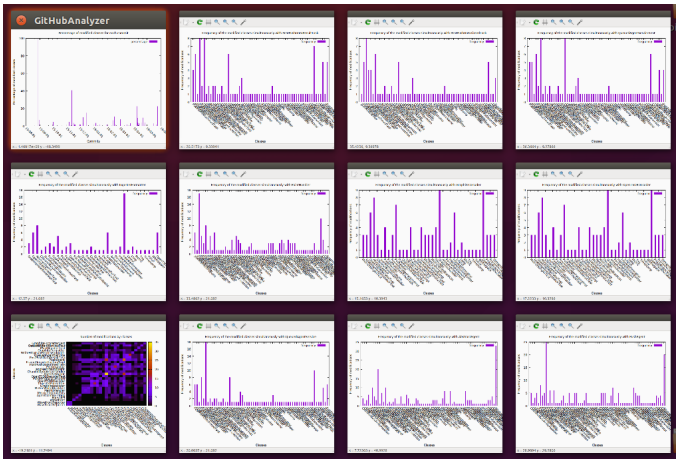

Fig. 3. Graphs produced by the analysis of *wavefrontHQ/java* repository. The left top graph is the result of the second statistic. The left bottom graph is the result, the heat map, of the third statistic. The other graphs are the 10 most modified classes of the repository.

IV. Testing

For the testing phase we used a machine with the following hardware configuration:

- CPU: Intel Core i7-4510U @ 2.00GHz x 4
- RAM: 8GB
- OS: Ubuntu 16.04 64-bit

The repositories of Java code chosen for testing are listed in Table I.

The results obtained from the analysis of the repository, about the execution time of the tool and the total number of classes to each repository are summarised in the graph of Figure 4.

For a better understanding of the results that the tool produces, we have prepared some graphs that show the results produced during the testing phase of the repository *wavefrontHQ/java*.

Fig. 4. Graph with the execution times in relation to the total number of classes of each repository analysed.

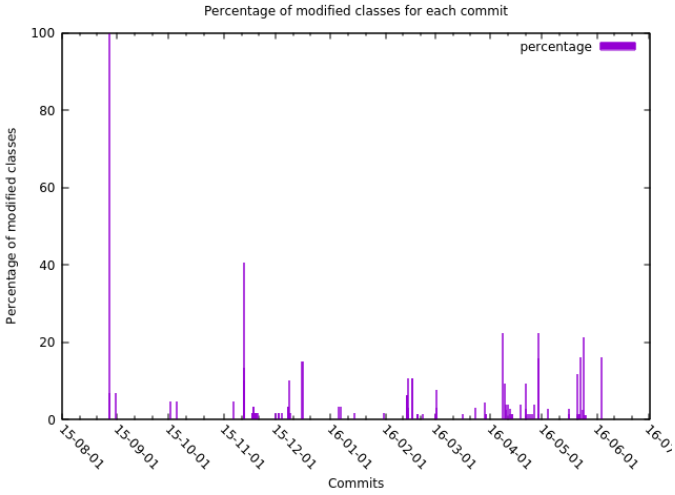

Fig. 5. Graph relating to the history of commits of the repository *wavefrontHQ/java*.

Figure 5 shows the time history of the commits of the repository that exceeds a threshold percentage of the classes modified for each commit, in this case the threshold is set to 0%. The graph shows that after the first modification of 100% of the classes, which corresponds to the first upload of classes in the repository, the repository has been changed to 40% of classes on one occasion during a commit dated November 2015 and it has not been modified for more than 20% from that moment forward.

Figure 6 shows that the class “GraphiteDecoder”, edited nine times during the commits to the repository, is strongly correlated to the class “OpenTSDBDecoder” which has been changed as often as the first was changed. In addition, there is a correlation, a little less strong, with classes “AbstractAgent” and “PushAgent”, and gradually more and more weak correlations with other classes. This type of chart is shown for the first classes based on the number of changes of the repository.

Figure 7 gives a heatmap for modified classes in relation to other classes. For a better visualisation, only a subset of classes

Repository	Number of commits	Last commit date	Number of classes	Execution time
wavefrontHQ/java [14]	181	13.65	80	10.430s
java-design-patterns [15]	1322	92.50	1029	555.537s
okhttp [16]	2535	33.33	583	462.374s
RxJava [17]	4715	8.99	2059	7189.825s

TABLE I
ANALYSED REPOSITORIES STATS.

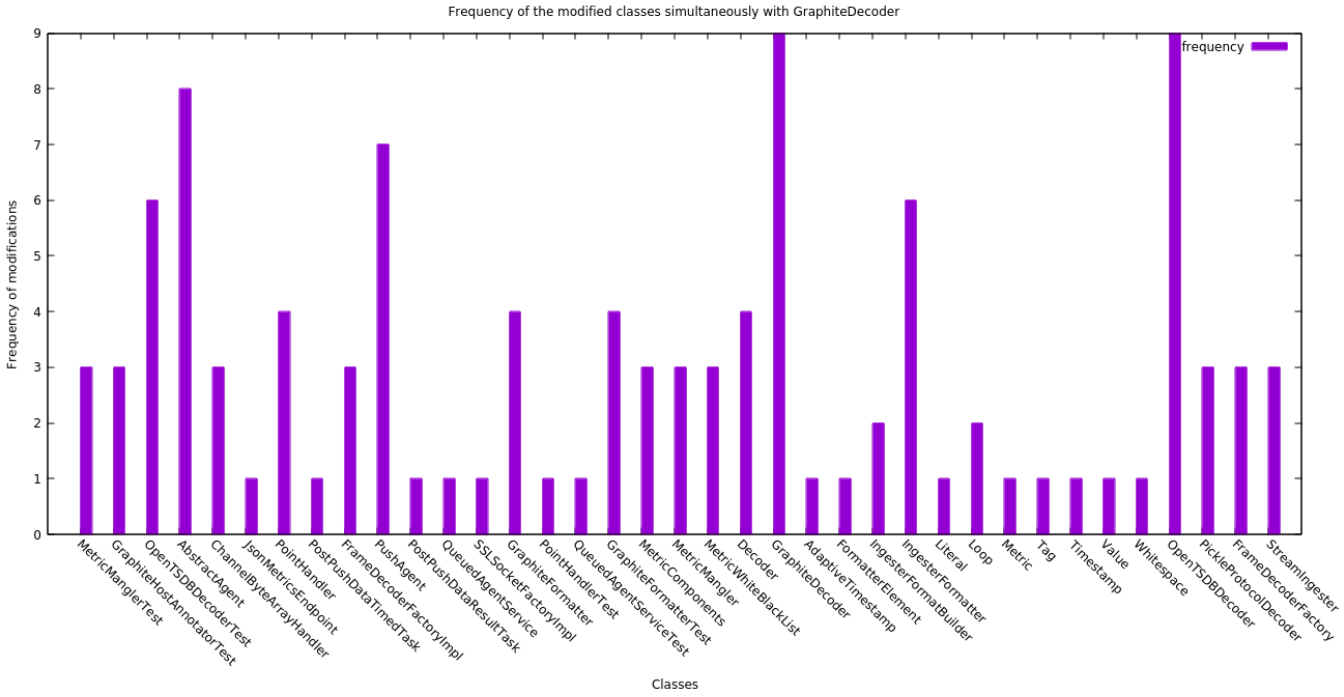

Fig. 6. Graph of the class "GraphiteDecoder" in the repository *wavefrontHQ/java*.

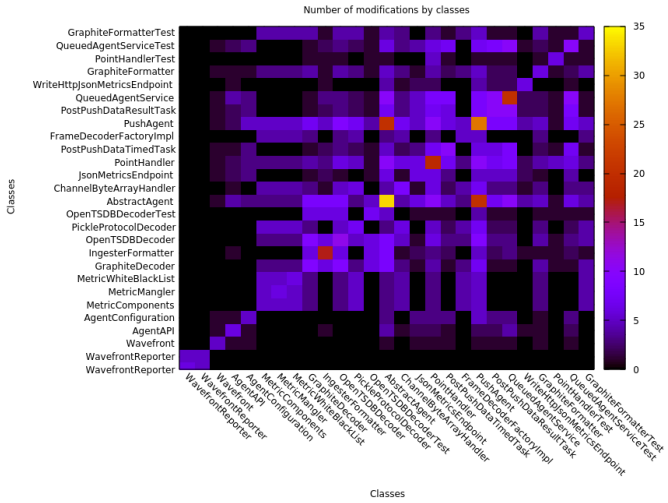

Fig. 7. Heatmap of the number of classes modifications in the repository *wavefrontHQ/java*.

has been shown for which the number of changes exceeds a certain threshold. The values on the diagonal represent the number of overall changes of the class and the values in the corresponding row (or column equivalently) are the number of changes of the other classes in the commits in which the first class has been modified. E.g. the row, or column, for class "AbstractAgent" let us see that it has been changed 35 times (color yellow) and along with it the class "PushAgent" was changed about 20 times (color close to red), while all other classes have been changed a number less than 10 times (colours from violet to black).

V. CONCLUSION

This paper has described the development of a tool that taps into GitHub repositories and extracts from them some relevant statistics for the commits of a system classes. The statistics, indeed, allow us to gain an insight into the software system, such as the classes strongly related to each other, since they were modified at the same time, most of times; and how many times the repository has been modified, by significant modifications, for the most part of code.

This work can be considered a preliminary, but essential, step towards developing further useful metrics, describing the work of the developers, or suggesting developers which parts of the system could be improved by means e.g. of refactoring techniques.

REFERENCES

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

[2] J. Kerievsky, *Refactoring to patterns*. Addison-Wesley, 2005.

[3] 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)*. Riva del Garda, Italy: ACM, March 2012, pp. 1866–1868.

[4] C. Napoli, G. Pappalardo, and E. Tramontana, "Using modularity metrics to assist move method refactoring of large systems," in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*. IEEE, 2013, pp. 529–534.

[5] R. Giunta, G. Pappalardo, and E. Tramontana, "Aspects and annotations for controlling the roles application classes play for design patterns," in *Proceedings of IEEE Asia Pacific Software Engineering Conference (APSEC)*, Ho Chi Minh, Vietnam, December 2011, pp. 306–314.

[6] A. Calvagna and E. Tramontana, "Delivering dependable reusable components by expressing and enforcing design decisions," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 493–498.

[7] R. Giunta, G. Pappalardo, and E. Tramontana, "Using Aspects and Annotations to Separate Application Code from Design Patterns," in *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2010, pp. 2183–2189.

[8] S. Cicciarella, C. Napoli, and E. Tramontana, "Searching design patterns fast by using tree traversals," *International Journal of Electronics and Telecommunications*, vol. 61, no. 4, pp. 321–326, 2015.

[9] E. Tramontana, "A design pattern for improving the performances of a distributed access control mechanism," in *Proceedings of AsianPlop*, Taipei, Taiwan, February 2016.

[10] F. Sokol, M. Zigmund, F. Aniche, and M. Gerosa, "Metricminer: Supporting researchers in mining software repositories," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013.

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] 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)*, 2016, pp. 593–601.

[13] T. Williams and L. Hecking, "Gnuplot," 2003.

[14] "Wavefront," 2016. [Online]. Available: https://github.com/wavefrontHQ/java

[15] I. Seppala, 2016. [Online]. Available: https://github.com/iluwatar/java-design-patterns

[16] "Square," 2016. [Online]. Available: https://github.com/square/okhttp

[17] "Reactivex," 2016. [Online]. Available: https://github.com/ReactiveX/RxJava