Classification of developpers based on git activity

Moret Jérôme
HES-SO Master
Lausanne, Switzerland
jerome.moret@master.hes-so.ch

Curty Pierre-Alain Charles
HES-SO Master
Lausanne, Switzerland
pierrealain.curty@master.hes-so.ch

Delessert Armand
HES-SO Master
Lausanne, Switzerland
armand.delessert@master.hes-so.ch

***Abstract*— The analysis of git activities provides useful information about the behavior of a team's developers. By simply analyzing the commits by hand it is difficult to give an idea of the performance of our developers. We propose in this article a procedure to obtain an effective visual result that will allow you to easily anticipate future problems in the development of an application.**

I. INTRODUCTION

To classify the developers, we proposed to be based on two metrics that are the churn and the throughput. Churn is understood to mean all lines of code that have been modified shortly after its publication. The throughput is simply the number of lines of code added. Based on these two axes, we would be able to obtain the position of a collaborator in this two dimensions' space. Exactly as Gitprime does in the following article [1].

II. GIT DATA EXTRACTION

To carry out this study, it is first necessary to build a dataset based on the repository' commits. It will be necessary to extract from these commits, all the files concerned to know if there have been additions, deletions or churns.

The git log command allows us to get this information. By parsing the output of the command, one can construct a dataset in the form of a csv file containing as header:

- Commit hash
- Commit name
- Author name
- Author email
- Commit date
- Filename
- Additions
- Deletions
- Churns

By creating this dataset, we offered the possibility of having a reusable data format for other analyzes. Or simply keep track of the old commits already dealt with, in an incremental analysis.

By playing on the parameters of the git log command, we get a result ready to be parsed. Here is the command used:

```
git log --pretty=format:
%h%x09%s%x09%ae%x09%an%x09%ai%x09%p --numstat
```

The numstat parameter activates filename listing with additions and deletions.

III. COUNTING CHURNS

Git considers a modification to be a deletion. Therefore, for each file with a deletions metric ≠ 0, there must be two conditions which confirm that one line is a churn:

1. The author is the same as the last modification
2. The difference between the date of this change and the last modification is smaller than a defined value. For example, 3 weeks.

To retrieve deleted (modified) lines number between this file and the last version, use the git diff command. Here is the command used:

```
git diff --unified=0
<parent_commit_hash>:<old_filename> <hash>:<filename>
```

The unified parameter helps to parse easily the modified lines.

The parameter <parent_hash>:<old_filename> specifies the last version before the modification. <old_filename> is a path to the *old* filename. Old because it may have been renamed or moved. We get this information at the git log moment field *filename*.

Example: {documentation => doc}/model.uml. The folder documentation has been renamed to doc.

Once we have the deleted lines number, we must compare the author and the date of the actual commit with author and date of each lines. If the conditions of a churn are confirmed then the deletion is considered a churn. This action can be performed using the git blame command. Here is the command used:

```
git blame -L<line_no>,+1 <parent_commit_hash> --
<old_filename>
```

The parameter –L<line_no>+1 indicate the line number followed by +1 to tell that we take one line.

IV. AGGREGATION OF THE DATASET

Now that we have built our dataset. We only have to sum the metrics additions and churns and group by author name. The result of additions sum gives us the throughput of a developer. The result of churns sum is a number of lines considered as churn, but to be an interesting measure, we must normalize this number of lines by the number of lines added. This gives us a percentage of lines rewritten in a short labs of time.

V. PRACTICE

In order to put this procedure into practice, we wrote a Python script that facilitates the parsing of git commands through the **GitPython** library.

Two Python scripts were written:

- *dataset.py* constructs the dataset presented in this article.
- *aggregation.py* performs aggregation by author.

You can find these two scripts on the following repository:

https://github.com/jermoret/T-SoftEng-ContributorsClassification

We can then retrieve the information emerged from the aggregation to make a graph with any spreadsheet.

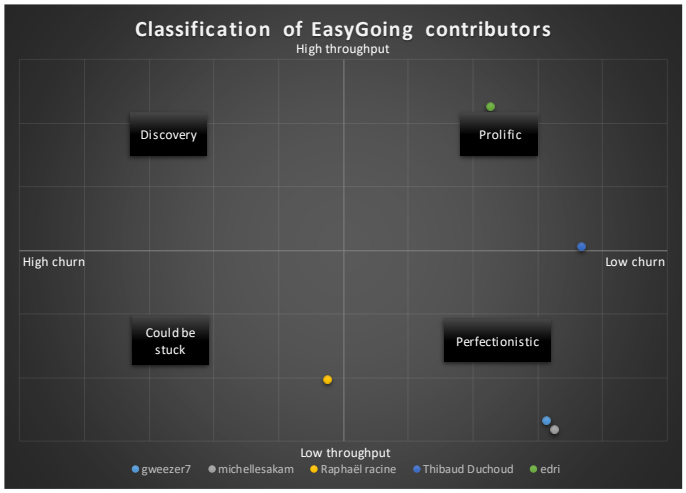

Figure 1 : test on a small repository (775 commits, 5 developers) https://github.com/edri/EasyGoing

Ideally we would like to have all our developers in the upper right corner. At the bottom right, we have developers who do not add much but who at least are not rewriting their little written code. At the top left, we will find developers who are doing tests with code which is legitimate according to certain phase of the project. Finally it is on the bottom left that we must inspect because we have there developers who produce few codes and who also constantly redo their code. A developer may need help because he could be stuck.

VI. INTERESTING CASE

We saw that we could analyze an entire repository in order to evaluate the performance of our team on the whole project but this is not always the most interesting case.

A much more interesting case appears in the use of this procedure within an agile team. Indeed, using the procedure described in progress or at the end of each iteration, we can target problems within our team. In this way you can react quickly by supporting the targeted developer or by quitting a false alert.

The script *dataset.py* contains a parameter –s, --since to do this work.

VII. FUTURE WORKS

At the moment this analysis is too complex, which results in an execution time that is often too long to analyze a large repository.

The construction of the dataset in **Figure 1** takes ~15min.

We will have to work on several points:

- Reduce the complexity of the algorithm.
- Try to parallelize the procedure.

In a final solution we can block execution of the program to a certain number of files events.

VIII. CONCLUSION

By exploiting the raw data of commits and working on well-defined metrics we see that we are able to measure the behavior of our developers through a git-based activity analysis.

By adding this measure to other measures based on more humane analyzes, you end up with a complete toolbox to help manage an agile team.

IX. REFERENCES

1] B. Thompson, «Prolific Engineers Take Small Bites — Patterns in Developer Impact,» 9 Mai 2016. [En ligne]. Available: https://blog.gitprime.com/check-in-frequency-and-codebase-impact-the-surprising-correlation/.