Capturing Software Evolution and Change through Code Repository Smells

Francesca Arcelli Fontana, Matteo Rolla, and Marco Zanoni

Department of Informatics, Systems and Communication
University of Milano - Bicocca
{arcelli,zanoni}@disco.unimib.it,matteo.rolla@gmail.com
http://www.essere.disco.unimib.it

Abstract. In the last years we have seen the rise and the fall of many version control systems. These systems collect a large amount of data spanning from the path of the files involved in changes to the exact text changed in every file. This data can be exploited to produce an overview about how the system changed over time and evolved. We have developed a tool, called VCS-Analyzer, to use this information, both for data retrieval and analysis tasks. Currently, VCS-Analyzer implements six different analysis: two based on source code for the computation of metrics and the detection of code smells, and four original analysis based on repositories metadata, which are based on the concepts of Repository Metrics and Code Repository Smells. In this paper, we describe one smell and two metrics we have defined for source code repository analysis.

Key words: Code Repository smells, Repository analysis, Repository Metrics, Code changes

1 Introduction

Code smells are well known in the literature [1], and researchers have been trying to automatically detect them and remove them through refactoring steps. Code smells point out symptoms of deeper problems and are based on the analysis of the code only. While searching problems in a system, not only a single code snapshot should be taken into account, but also its history that will eventually result into code smells or other problems or symptoms of problems. Analyzing code changes is a natural way to track developers behavior. Version Control Systems (VCS) play a huge role in software development. There is no safe way of merging the work of several developers without using a VCS and furthermore no developer nowadays would work on a project of some importance, size or

value, without being able to revert changes when things get out of hand. VCSs allow to store and expose many data on the contents, authors and times of the changes applied on a project repository.

In this paper, we show some ways these data, or part of them, can be used to extract symptoms of more deep-rooted problems. Since these symptoms are based on repositories and for the analogy with Code Smells, they have been called *Code Repository Smells*. In particular we focus our attention on a Code Repository Smell, which we called *Code Bashing*. This smell refers to the situation in which changes made by several developers on several versions of a specified file gather in a narrow portion of the file itself.

Moreover, we have defined two new metrics, for data gathered from repositories, called the *Repository Stability* and the *File Volatility* metrics. Repository Stability is based on the concept of file closures and represents, at a given moment, the ratio between stable files and those that will be subjected to further modifications. File Volatility express how much the content of a file, or a portion of it, changes in relation to the number of its versions.

For our analyses, we have developed a tool, called VCS-Analyzer, that allows retrieving data needed for software analysis harvesting system repositories. It does not rely on any particular VCS, but instead it produces a model that abstracts and encapsulates all the data retrieved from such systems, guaranteeing complete independence. VCS-Analyzer needs only the identifier of the repository to be analyzed, and takes care of the retrieval of metadata or files, depending on the analysis to perform. It has been designed to allow simple plugging of new analyses and other VCS, by decoupling the different aspects of repository crawling. The tool currently supports Git and SVN, which are the most used VCSs in the open source community, but others can be integrated if the need for them arises. VCS-Analyzer has been used to perform different analyses on many systems, e.g., JUnit, ElasticSearch, the Linux kernel. The detailed description of VCS-Analyzer and the analyses performed can be found in the Thesis of M. Rolla [2]. Currently the supported analyses exploit the detection of code smells, the computation of many metrics, change sizes and number of changes, as well as the information captured through the Code Repository Smell and Metrics defined in this paper.

The paper is organized through the following sections: in Section 3 we introduce the Code Bashing smell. In Section 4 and Section 5 we introduce and describe the two new Repository Metrics. Finally, in Section 6, we conclude and outline some future developments.

2 Related Work

At the best of our knowledge, the literature does not report *Code Bashing* smell, as defined in this paper, and the same holds for the identified Repository Metrics.

For what concerns the different analysis on the evolution of software repositories, as those we can perform with VCS-Analyzer, many works have been proposed in the literature, as the papers in the Proceedings of Mining Software Repositories Conferences [3] and many other works [4, 5, 6, 7].

While for what concerns systems similar to VCS-Analyzer, different works have been proposed in the literature, as for example *Churrasco* [8], which provides software evolution modeling, analysis and visualization through a web interface, or *Kenyon* [9], a system designed to facilitate software evolution research by providing a common set of solutions to common problems.

The aim of VCS-Analyzer is different from many other tools, in particular it implements the computation of code related metrics and the detection of code smells as well, and its main focus is supporting the software assessment process analyzing data exposed (and thus available), by the version control systems rather than the code itself.

3 Code Bashing Smell

Code is rarely completely right at first writing: it undergoes several changes and optimizations, which are natural during its evolution. When a portion of code keeps changing frequently it may point out a deeper problem in the system structure or design. Nevertheless, when developers reiteratively edit the same portion of code, it is sign of a problem: e.g., either the requirement specifications were not exhaustive and subject to frequent change, or the code is too complex for the developer to be fully understood. Moreover, once a piece of code is considered stable enough, changes involving it should be in a limited number, following the principle of single responsibility [10] which states that every class should have a single reason to change, because it has a single responsibility. Even in agile methods, where changes are normal and welcome, an excessive change to the same piece (or single line) of code can be suspicious. Clearly, depending on the repository branch and conventions, the evaluation of the amount of changes can have different interpretations. A local development branch will be by far more unstable than an official stable/release branch. It often happens that local branches are used to experiment different solutions, and files get almost totally changed from time to time. On mature projects, instead, there is usually

a branch which receives only tested and approved code. When too many changes are made to this kind of code, it is often the sign that a problem happened.

We can think to assign to the code a value, able to express how much it has been exposed to changes. The idea is to assign to each line of code a score, called *Changing Intensity*, representing how many times the line has been touched by some changes. The higher the score, the more a line participated to changes. The algorithm we implemented in VCS-Analyzer populates an array of *Changing Intensity* for every text file in the repository. To track change positions and text, the algorithm analyzes the differences between the same file in consecutive versions, using the patch texts provided by the VCS. All major VCSs embrace the unified diff format. The lines of a diff can be grouped into three categories:

- *Neutral blocks*: composed of lines serving as context and not taking part in changing the file.
- *Additive blocks*: composed of lines marked with the plus sign (+) by the diff; these are the lines added in the new version of the file.
- *Subtractive blocks*: composed of lines that are marked by the minus sign (-); they represent the lines that will not be present in the new version.

The algorithm splits the diff text and the file in blocks, reassembling them in a way that the result is a sequence of blocks with additive, subtractive and neutral blocks. Then the vector containing the changing intensities of the previous version of the file is updated accordingly to the changes represented by the different blocks. Changes in the unified diff format are in form of additions and deletions only. Edits are represented by the deletion of lines followed by the addition of the former lines, incorporating the change. The way the scores are assigned to lines is guided by a simple score assignment schema we propose.

The idea behind the score assignment schema is that the score of every line should represent how many times the line has been changed. Following this criterion we manage different cases:

- isolated additive block: inserted lines at the end of file get score 1, the others get the score of the line at the same position plus 1;
- isolated subtractive block: deleted lines get removed, and their score is also removed; the lines before and after the block (if existing) get their score plus 1, witnessing the code removal;
- subtractive block followed by additive block, representing a change: depending on the size of the two blocks we consider three cases:
  - same size: lines get the score of the substituted lines plus 1;

- addition longer than deletion: the scores of the deleted lines, plus 1, are passed to the first added lines; remaining lines get score 1;
- deletion longer than addition: all added lines except the last one get the respective deleted line score plus 1, while the last gets the maximum score of the remaining deleted lines plus 1.

In the following, we report an example of the application of the score assignment schema. For space reasons, we focus on the last case (modification with more deletion than addition), which is the more complex and less intuitive one. In Listing 1, Listing 2 and Listing 3, we show a file before and after a change, and the diff file for the change. Each line of the file has an associated score, before and after the change. For the sake of simplicity, in the following listings, changes are represented only in the form of a single line or a few lines, but the real implementation obviously uses blocks instead.

Listing 1. Original file

```
1 public class Test {
4     private String key;
4     // first line comment
1     // second line comment
1 }
```

Listing 2. Resulting file

```
1 public class Test {
4     private String key;
5     // single line comment
1 }
```

Listing 3. Difference between the two files

```
@@ -1,5 +1,4 @@
 public class Test {
     private String key;
-    // first line comment
-    // second line comment
+    // single line comment
 }
```

4 File Volatility Metric

File Volatility is the first Repository Metric we defined[1]. At every moment during its life, each file has associated a Changing Intensity vector, calculated with the method described in section 3.

We can then define the *volatility* of a file respect to its evolution, as the ratio between the maximum value in the Changing Intensity vector and the number of changes to the file since its creation. This value represents how frequently a piece of code has been changed during the evolution of the containing file up to a given moment. A value close to 1 that means in the file there is a portion of the code that is kept being involved in changes from the beginning until the last version. This behavior is clearly not desirable, or at least points to a peculiar situation. Figure 1 represents the plot of the *File Volatility* of the file `pom.xml` in the ElasticSearch[2] project. In an ideal situation, the value of the metric decreases in time, starting from 1. When a file is created its File Volatility is 1, and at each change it keeps close to 1 if the changes are applied to the same region of code. Otherwise, the value of the metric tends to be lower. In the example, the value of the metric quickly decreases to smaller values, meaning that the changes involved different regions of the file. In that particular file, the only line of code receiving a high score is the one representing the version of the system. In fact, the same line has been edited every time the system changed its version number.

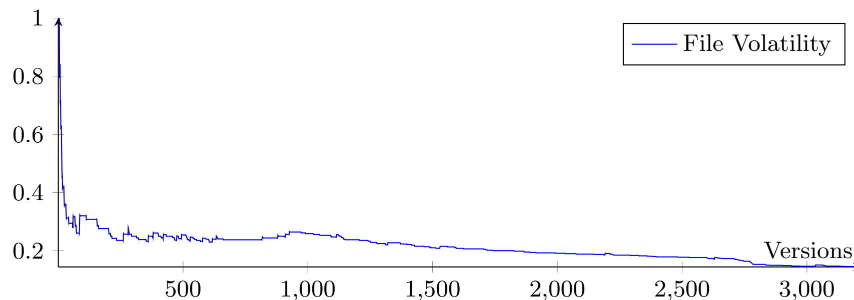

Fig. 1. Evolution of the File Volatility metric

Code refactoring techniques like *Move Method*, *Extract Method* and *Extract Class* can greatly affect the value of the metric. Consider for example the case

[1] At the end of the section, we outline the differences with other existing metrics like code churn.

[2] `https://github.com/elasticsearch/elasticsearch`

of problematic code in a method, which is subjected to many changes for several reasons and was not even supposed to be in that particular class. It could be a case, e.g., of a *Feature Envy* method. To remove such smell, the refactoring to apply could be to move that method elsewhere in the code base. If that method had, into its body, lines with the highest values of Changing Intensity in the file, then its removal would dramatically drop the volatility metric value. The same statement holds w.r.t. every refactoring technique that implies the removal of a considerable portion of the code, as well as deleting code for other reasons.

File Volatility is different from other existing metrics measuring code changes. Two widely investigated change measures are the number of file changes and code churn [7]. File Volatility expresses a different measure than the number of file changes. In fact, it summarizes the changes of the single lines, relative to the number of file changes. For example, a file receiving 10 new lines in 10 versions, one line per versions, has a number of changes value of 10. Its File Volatility, instead is 1/10 (assuming no other lines have ever been changed). The code churn (in its simplest form) for the same file will be (absolute form) 10, or (relative form) the average of 1/LOC for each of the 10 versions. File Volatility is a measure evaluating the peaks of line changes in files, while code churn is related to the size of every change made to the file, without recognizing the identity of the single lines.

5 Repository Stability Metric

The second Repository Metric we defined is Repository Stability. Following the principle of single responsibility, when a piece of code reaches enough maturity, the chances of it being changed are extremely low. M. Feathers defined that a class can be considered closed [11] at time t if no further modifications will happen from t to present. The same concept can be extended to files: a file can be considered closed when no further development is done on it from a version to the present one. When a file reaches enough maturity, there is high probability that it will not be subjected to future changes. Given this assumption, tracked files can be grouped in *active files* and *closed files*. To give a graphical immediate representation of the concept of *file closures* described above, we report, as an example, the values computed on JUnit, chosen for its long change and development history. JUnit's repository[3] is managed by Git, after the migration from *CVS*. Figure 2 shows the amount of files involved in every single commit.

[3] `https://github.com/junit-team/junit`

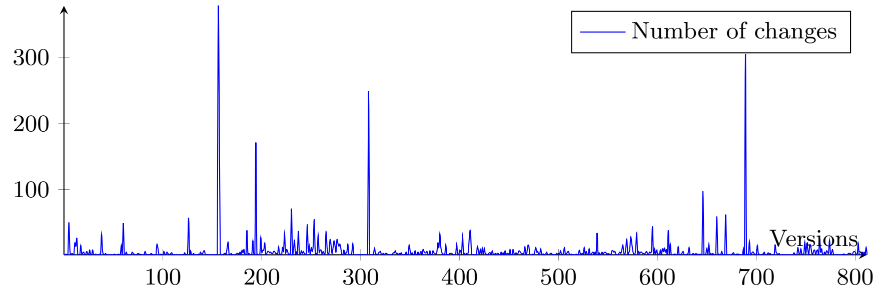

Fig. 2. Number of files changed each version

We can see that there are few commits that really stand out from the average and especially the last one, as we will see later, has a remarkable impact on the value of the metric. In Figure 3 we can see a comparison of the evolution of total and closed files.

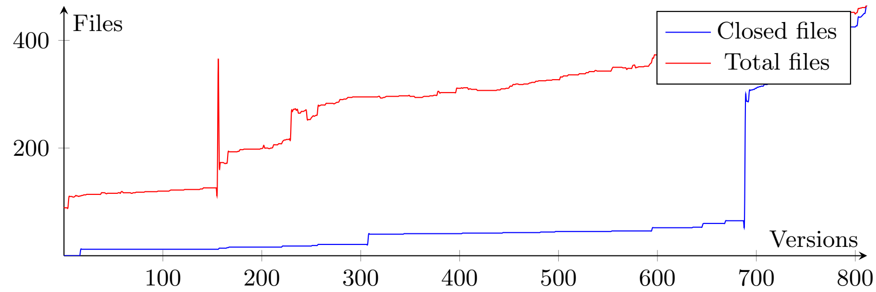

Fig. 3. Evolution of closed/total files

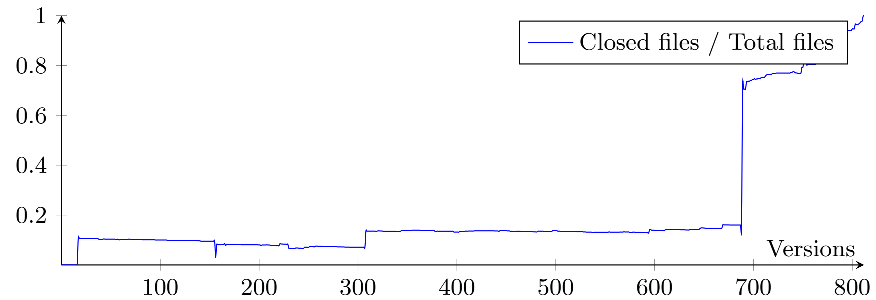

Fig. 4. Evolution of the Repository Stability metric in JUnit

The commit near the end represents the reason why the metric keeps low values for almost the entire life span of the repository, except for the last two hundreds versions. The development team decided to apply new coding conventions to the code base. This decision resulted in a huge number of files involved into the change. Hence, only the files not interested by the new standards could be considered closed. Just after the commit, the number of closed files suddenly increases and keeps increasing until the end of the timeline. In the last commit, the number of closed and total files are the same. This is due to the fact that, by construction, the odds of finding a file contained in a change set decrease proportionally with the progress in the history of commits. At last commit, none of the files can be found at a later stage. The desired development behavior is to focus on single functionalities and then move to others when the implementation is mature enough. By looking at the Repository Stability evolution graph, one can immediately judge if developers are following this principle. The Repository Stability evolution graph for a given interval, should keep growing as time advances and the gap between active and closed files should be as narrow as possible. Even in agile development environments with short release cycles and incremental refactoring, there is a time when code has to stop changing and become stable. Obviously, code can't be mature from the start, but it should be at some time in the future.

6 Conclusions and Discussions

In this paper, we have described a new Repository Smell, highlighting code regions that received more attention than others. We defined also two Repository Metrics, giving a quick overview of the level of maturity of single files and of an entire repository. In particular, File Volatility assigns to each file a score, telling how much changes are concentrated on particular lines, revealing code regions that needed more attention than others, and could need more in the future. Repository Stability, instead, can summarize the portion of repository which did not need to be changed since an instant in its development history; repositories where most files change over time can suffer from organizational or design issues, so this metric can reveal potential quality problems.

This information can be exploited for software maintenance and quality assessment. For example, the selection of a third party open source component can be aided with measures characterizing the maturity of the project, as well as other issues related to its development process.

We already analyzed different projects, i.e., Mozilla Rhino, JUnit, ElasticSearch, RSpec (core package), Tomcat, ION, and the Linux kernel. The results of these analysis are available in a web page[4] that we will keep updated with results obtained on new projects. In the past, we focused our attention on code smell detection and assessment [12]. Now we aim to focus on finding smells tied to software evolution and repository analysis, to extend our experimentation with the new Repository Smells and discovering new ones. Moreover, through VCS-Analyzer we intend to perform different empirical analysis for assessing the quality of software projects, basing on their development history.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
2. Rolla, M.: Empirical analysis for software assessment. Master's thesis, University of Milano-Bicocca, Viale Sarca, 336, Milano, Italy (January 2014)
3. Zimmermann, T., Penta, M.D., Kim, S., eds.: Proc. 10th Working Conference on Mining Software Repositories (MSR '13), San Francisco, CA, USA, May 18-19, 2013. In Zimmermann, T., Penta, M.D., Kim, S., eds.: MSR, IEEE / ACM (2013)
4. Peters, R., Zaidman, A.: Evaluating the lifespan of code smells using software repository mining. In: Proceedings of the 16th European Conference onSoftware Maintenance and Reengineering (CSMR 2012). (2012) 411–416
5. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maint. Evol. **19**(2) (March 2007) 77–131
6. Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S.: Mining version histories to guide software changes. **31**(6) (June 2005) 429–445
7. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International Conference on Software Engineering (ICSE 2005). (May 2005) 284–292
8. D'Ambros, M., Lanza, M.: Distributed and collaborative software evolution analysis with churrasco. Science of Computer Programming **75**(4) (2010) 276–287 Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).
9. Bevan, J., Whitehead, Jr., E.J., Kim, S., Godfrey, M.: Facilitating software evolution research with kenyon. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13), Lisbon, Portugal, ACM (2005) 177–186

[4] http://essere.disco.unimib.it/VCSAnalyzerResults.html

10. Martin, R.C.: Chapter 9 — SRP: The Single Responsibility Principle. In: The Principles of OOD. objectmentor.com (February 2002) `http://www.objectmentor.com/resources/articles/srp.pdf`.
11. Feathers, M.: Working Effectively with Legacy Code. Robert C. Martin Series. Pearson Education (2004)
12. Arcelli Fontana, F., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. J. Object Technology **11**(2) (Aug 2012) 5:1–38