Does the Act of Refactoring Really Make Code Simpler? A Preliminary Study

Francisco Zigmund Sokol[1], **Mauricio Finavaro Aniche**[1], **Marco Aurélio Gerosa**[1]

[1] Department of Computer Science
University of São Paulo
PO Box 66.281 - 05.508-090 - São Paulo - SP - Brazil

francisco.sokol@usp.br, {aniche, gerosa}@ime.usp.br

Abstract. *Refactoring is the act of changing software code, usually to improve internal code quality, without changing its external behavior. Soetens and Demeyer (2010) analyzed one software and showed that code refactoring did not imply in better result for code quality metrics. In this work, we extend Soetens and Demeyer study, mining data from 256 software projects from Apache Software Foundation, using MetricMiner, a web application focused on supporting mining software repositories studies. The quantitative analysis showed that refactoring indeed does not decrease Cyclomatic Complexity. On the other hand, the qualitative analysis showed that a refactoring tends to improve code in terms of readability and maintainability.*

1. Introduction

Refactoring is the practice of improving code quality without changing the external behaviour of a software system [5]. If applied correctly, refactoring improves the maintainability of a system, without affecting its funcionality. Thus, it is expected that refactoring practices lead to better results in terms of code quality metrics, such as Cyclomatic Complexity [13], altough this metric measures only one code quality attribute, namely, complexity.

To understand the effect of refactorings on the quality of a system, the study published by Soetens and Demeyer [11] analyzes the development history of PMD[1], an open source system, relating the commit messages of its SCM with the evolution of the Cyclomatic Complexity metric.

In this paper, we reproduce and extend that study, analyzing the history of 256 different open source Java projects from the Apache Software Foundation. We also propose a more efficient strategy in mining those repositories using a web application designed to support mining software repositories studies, namely MetricMiner. We analyzed all the 256 java projects that exist in Apache Software Foundation and are available via Git interface.

We found that refactoring practices do not reduce Cyclomatic Complexity significantly, confirming the results of the replicated study in a larger dataset. However, when analysing a small subset of commits, we noticed that refactorings tend to improve system code in terms of readability and maintainability. In addition, we present some exploratory

[1]http://pmd.sourceforge.net/. Last access on April, the 25th, 2013.

graphs that suggest different patterns of the evolution of Cyclomatic Complexity among projects.

This paper is structured as follows. In Section 2, we briefly describe what is refactoring and present background works that inspired our study. In Section 3 we report the method used to extend the original study. In Section 4, we present the results obtained. In Section 5, we analyze our results and compare them with Soetens' study. In Section 6, we discuss the possible threats to the validity of this study. Finally, in Section 7, we show our conclusions and further work.

2. Literature Review

In this section we briefly introduce the definition of code refactoring, present some related studies in refactoring and code quality and describe Soetens and Demeyer's study.

2.1. The Act of Refactoring

Refactoring is used to improve code quality. It aims to make the code more readable and, therefore, more maintainable. The practice of refactoring is made by small code modifications that preserves its behavior, but improves the overall internal quality of the software.

Many refactoring techniques are documented in Fowler's book [5] and usally adopted in practice and implemented in the integrated development environments.

Refactoring may also improve the extensibility of the code. As described in [6], many design patterns can be applied in code refactoring. Refactorings such as removing conditionals with polymorphism (applying the Strategy pattern, for example) boost up the code reuse and also improve its maintainability. With all that being said, it is expected that, when a developer affirms that s/he has done a refactoring in the code, the code gets better, and its general complexity decreases.

In this paper, we focus in measuring the effect of refactoring only in terms of code complexity, which is only one attribute of code quality. To measure this attribute we used Cyclomatic Complexity metric. This metric was defined by McCabe [13], in 1976. The metric measures the amount of the different possible logical paths a method can have. Each time there is a branch (for example, an *if, for, while, switch, etc*), the number of the complexity increases by 1. All methods have at least 1 as its Cyclomatic Complexity.

To illustrate the effect of a refactoring over the Cyclomatic Complexity, consider the following excerpt of a Java class:

```
public void Employee {
    ...
    public void pay(double value) {
        double factor = 1.0;
        if (isManager()) {
            factor = 0.9;
        }
        this.money += value * factor;
    }
    public void payBonus(double value) {
        double factor = 1.0;
        if (isManager()) {
            factor = 0.9;
        }
        this.bonus += value * factor;
    }
    public void payOvertime(int hours) {
        double factor = 1.0;
        if (isManager()) {
            factor = 0.9;
        }
        this.money += hours * getHourValue() * factor;
    }
    ...
}
```

Since `pay` method contains one `if` statement, its code have two possible paths of execution (varying on the result of `isManager()` call), resulting in a value of 2 of McCabe's Cyclomatic Complexity. Similarly, `payBonus` and `payOvertime` methods also have 2 of Cyclomatic Complexity, therefore, the Cyclomatic Complexity of this class is 6 (considering only the excerpt presented).

A simple refactoring can be applied to this class, namely, "Extract Method"[5], which consists in the extraction of a code fragment to improve readability and possibly removing duplication. In this case we can clearly remove some duplicated code:

```
public void Employee {
    ...
    public void pay(double value) {
        this.money += value * calculateFactor();
    }
    public void payBonus(double value) {
        this.bonus += value * calculateFactor();
    }
    public void payOvertime(int hours) {
        this.money += hours * getHourValue()
           * calculateFactor();
    }
    private double calculateFactor() {
        double factor = 1.0;
        if (isManager()) {
            factor = 0.9;
        }
        return factor;
    }
    ...
}
```

The refactored class totalize 5 of Cyclomatic Complexity, since `pay`, `payBonus` and `payOvertime` all have 1 of Cyclomatic Complexity and the new extracted method, `calculateFactor`, have 2.

2.2. Related Work

There are a few studies in the literature on the effects of refactoring in code quality. There are still no conclusions about it, as some studies affirm that refactoring improves code quality, others say that it brings negative impact to the code, and others even state that it does not make any difference at all.

Alshayeb's study [8] investigates the effect of the act of refactoring in a few quality attributes, such as adaptability, maintainability, understandability, reusability, and testability. He was not able to find any trends on the effects of refactoring in those attributes, and he concludes that it is not possible to affirm that refactoring improves software quality.

In contrary to Alshayed, Du Bois et al. [1] proposed a controlled experiment with students in order to study the effects of refactoring on the understandability of the code. They found out that refactoring can improve program comprehension.

A few studies, besides ours and Soetens', used mining software techniques to discuss the effects of refactoring. As an example, Stroggylos and Spinellis' [12] mined data from three open source projects and studied the effect of refactorings in different quality metrics. Considering all metrics, they found that refactorings actually decrease the internal quality of the software. However, Geppert et al. [3] studied the modifications history of a industrial communication system and showed that refactorings in that project caused a positive impact on changeability and defect density of that software system.

2.3. Soetens and Demeyer's Study

Soetens and Demeyers (2010) analysed the relation of refactoring and the metric of Cyclomatic Complexity, used to measure the total complexity of a system. They take the sum of the value if the metric for each method from each class as the measure of the complexity from the whole system.

Soetens and Demeyers analyzed 776 versions extracted from the version control system from the open source project PMD. To accomplish that analysis, the authors used SVNKit – an java API to access SVN repositories – and Eclipse Metrics – an Eclipse IDE plugin that calculates different code metrics. They used SVNKit to to extract the code revisions from the repository, and Eclipse Metrics to calculate the Cyclomatic Complexity.

With support of those tools, the authors developed a second Eclipse plugin to automate the process of loading the code revision, calculate the metrics, and store the results into an XML file. After that, the XML files were processed and the metric results of each revision were associated with the corresponding commit message.

Their paper presents a chart of the evolution of the Cyclomatic Complexity. In Figure 1, we display that chart, showing that the complexity of a system tends to increase over time, in an evidence of the second law of software evolution [7]: the complexity of an evolving program increases during its development history, unless work is done to maintain or reduce it.

The authors classify the commits in two categories: those that contain "documented refactorings" and those that do not. A "documented refactoring" is a commit that

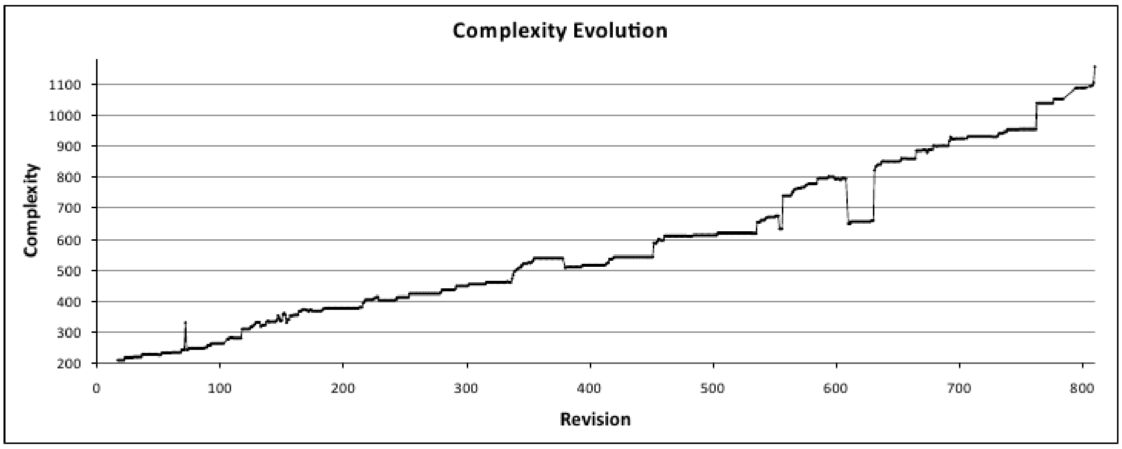

Figure 1. Chart of the Cyclomatic Complexity over time published in [11]

	Decrease CC	Equalized CC	Increased CC
Documented Refactoring	14	7	12
No Refactoring Documented	27	580	136

Table 1. The findings in Soetens and Demeyer work.

has an associated message containing words such as "refactoring", "refactored", "refactor", and so on. The approach of relating an act of refactoring by reading the commit message was first presented by Ratzinger et al. [4].

With this classification, the authors analyze the effect of Cyclomatic Complexity on the commits from each category. In Table 2.3, the results of that analysis are presented.

With those results, the authors conclude that refactoring rarely affect the complexity of a system: "*we discovered that refactoring practices rarely affect the Cyclomatic Complexity of a program*." The authors analyze the processed commits one by one and justified this counterintuitive conclusion with three main possible reasons:

- Few refactorings involved removing code duplication.
- In many commits, developers created new funcionallity together with the refactoring of some part of the code, increasing the overall complexity apart from the refactoring, supporting the conclusions made by Murphy-Hill et al [9].
- Most of the refactorings only affected small part of the code, involving simple refactorings such as moving methods and variables instead of more complex modifications such as extracting classes, replacing conditionals with polymorphism and so on.

3. Replication Protocol

Our intention in reproducing the study described previously was to extend the original work. Mining data from a larger set of software projects of different application domains would possibly find different patterns of the evolution of Cyclomatic Complexity than the one presented by Soetens and Demeyer. Furthermore, we wanted to propose a more effective methodology of analysing the evolution of code metrics, supported by MetricMiner[2], a web application focused in supporting Mining Software Repositories research.

Currently, MetricMiner has data and software metrics mined from the development history of all Apache Software Foundation Java projects. This data is available for consulting through SQL queries submitted through a web interface. Therefore, a SQL query was executed through MetricMiner to extract the calculated Cyclomatic Complexity from each Java class from the whole history of the projects stored in its database. Together with the metric value, the query also extracted the name of the file, name of the project, date and commit message associated.

The result of this query can be found in MetricMiner web interface[3]. With these results, a simple auxiliary Java program was developed to process the query result and measure the effect of refactorings on the total Cyclomatic Complexity for each project [4]. This auxiliary program groups the data extracted by project, groups files modified in a same commit, and navigates on the whole commit history, measuring the variations over the total Cyclomatic Complexity of each project. To verify if a commit contains "documented refactorings" (using the same approach proposed in the original study), we used Apache Lucene[5] to process the commit message, splitting the text into tokens and performing the normalization and stemming (two common information retrieval techniques). Since the method implemented used pre-processed data mined by MetricMiner, we were able to analyze a larger data set than the original study. A total of 256 projects and approximately 250,000 commits were analyzed in this process, which took less than an hour to accomplish.

4. Findings

In Table 4, we exhibit the results of the documented refactorings over the complexity of the whole set of projects analyzed. Most of the refactorings, in fact, increase the cyclomatic complexity. However, the rate of commits that decrease the complexity is larger in the set of the documented refactorings: approximately 23% of the documented refactorings decrease the complexity while only 12% had the same effect among the other commits.

	Decrease CC	**Equalized CC**	**Increased CC**
Documented Refactoring	1504	1603	3230
No Refactoring Documented	30145	99580	121239

Table 2. The findings of our replication study.

Analyzing the effect of the commits per project, we observed that the documented refactorings had a slightly better effect over the cyclomatic complexity than the other

[2]http://www.metricminer.org.br/. Last access on April, the 25th, 2013.

[3]http://metricminer.org.br/query/1. Last access on April, the 25th, 2013.

[4]https://github.com/csokol/refactoring-cc/. Last access on April, the 25th, 2013.

[5]http://lucene.apache.org. Last access on April, the 25th, 2013.

commits. In Figure 2, we show two scatter plots showing the percentage of commits that decreased Cyclomatic Complexity by each project's total commit count. Each dot represents a project and only the projects with documented refactorings were considered.

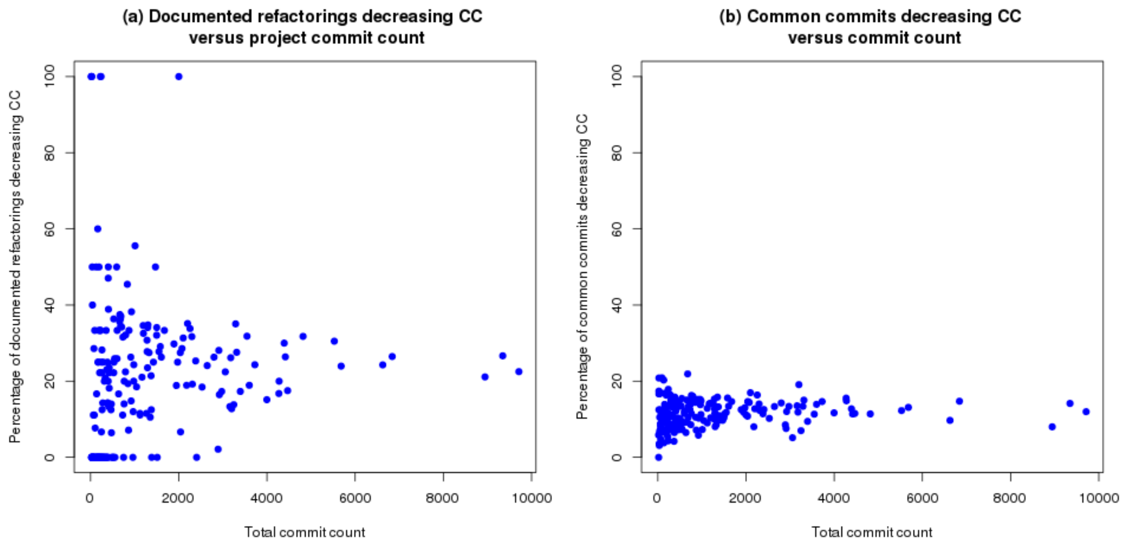

Figure 2. Percentage of commits decreasing CC

For the scatter plot *(a)* only the set of documented refactoring commits were considered, so each point exhibits the percentage of commits that decreased complexity among the group of documented refactorings. In scatter plot *(b)*, only the set of common commits were considered. Visually, we can see that the percentage of refactorings that decreased complexity is slightly higher than the common commits. The median of the dataset *(a)* is 20%, while for the dataset *(b)* is 11.38%.

4.1. Case Studies: Apache Camel, Ant, and Tomcat

Among the whole set of projects analyzed, we found some different patterns of the evolution of Cyclomatic Complexity. In Figures 3, 4 and 5, we show the evolution of the complexity of the Camel, Ant, and Tomcat projects.

The chart of Apache Camel presents a pattern similar to the PMD project mentioned previously. Its Cyclomatic Complexity starts from a low value and keeps growing with a practically fixed growth rate, resulting in an almost linear graph.

The history of Ant also starts from a low complexity. However, along its development there are "jumps" of the complexity caused by large insertions or removals of code from the repository. For example, in 4(**a**) there is a valuable increase of the total complexity that is justified by a large insertion of code from the older project repository, which is described in the author's message of this commit: "*Add in a clone of the main ant source tree so that it can undergo some heavy refactoring*."

The history of Tomcat, unlike the previous projects analyzed, starts from a high complexity from its first commit. This observation was caused by the fact that Apache Tomcat was a legacy code project. Because of that, its first commit is the initial import of the code of the older repository, which justifies the higher initial Cyclomatic Complexity.

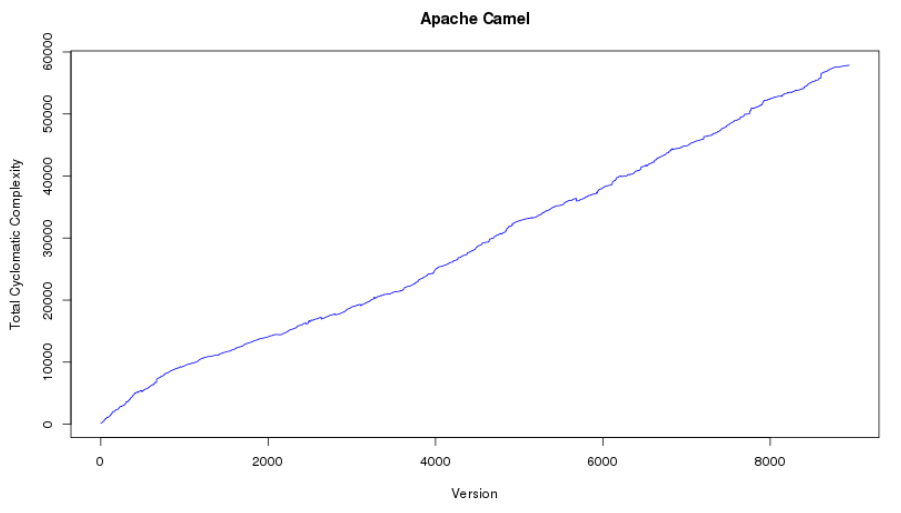

Figure 3. Complexity evolution of Camel project

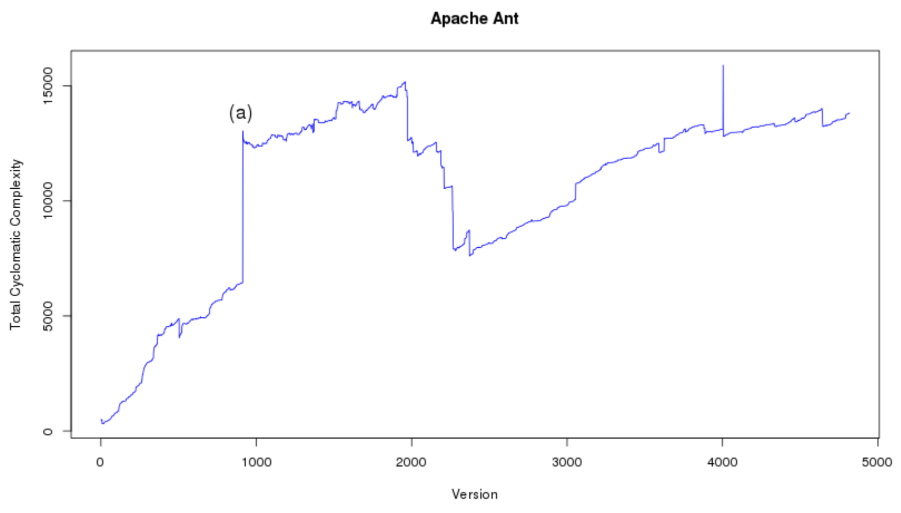

Figure 4. Complexity evolution of Ant project

In general, the three projects increase their complexity along their development history, confirming the second law of software evolution and corroborating with the study of Soetens and Demeyer.

4.2. Qualitative Analysis

We randomly selected 50 commits from the 3 studied projects, in which 25 of them were refactorings that increased CC, and the other 25 were refactorings that decreased CC. We then manually analyzed each of them and classified the change that actually happened in that commit. A commit could be classified in more than one category[6].

In Table 3, we present the categories we found for the commits that decreased CC. One can notice that 68% of them decreased CC by removing code. It means that

[6]The selected commits, as well as the categories can be found in this link: http://bit.ly/11IgJac. Last access on April, the 25th, 2013.

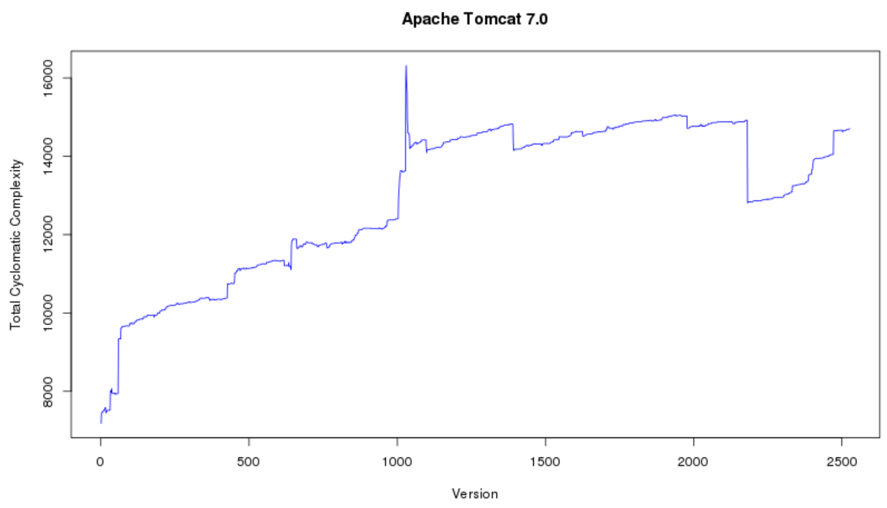

Figure 5. Complexity evolution of Tomcat project

part of the implementation or even full methods were deleted. Naturally, the overall CC decreases. In 24% of them, we also detected full classes being removed. However, in 20% of them, we detected an *Extract Class* refactoring. It means that a complex method (or a couple of methods) were moved to a new class, in a more elegant way. *Extract method* and *Extract Interface* were also detected in the manual inspection.

Category	**Quantity (Percentage)**
Code Removed	17 (68%)
Extract Class	5 (20%)
Extract Method	1 (4%)
Class Removed	6 (24%)
Extract Interface	1 (4%)

Table 3. Distribution of the 25 Commits that Decreased CC.

In Table 4, we present what happened in those refactorings that actually increased CC. In 60% of them, new features was added to the project and, therefore, increased complexity. On the other hand, we were able to identify many other refactorings, such as Move Method (4%), Rename Method (8%), Extract Super Class (16%), Extract Class (28%), and Pull Up Method (12%), which theoretically improves the quality of the code (in terms of readability and maintainability). These refactorings can or cannot increase CC. As an example, suppose a *Extract Super Class* refactoring, which removes duplicated code from two or more classes. In this case, CC would decrease.

A divergent point is that the "Rename Method" refactoring increased the cyclomatic complexity. It can be explained by the fact that a commit can contain more than one refactoring at once (Rename Method and Functionality Added, for example). So, although the "Rename Method" refactoring appears on the list, it does not increase the

Category	**Quantity (Percentage)**
Functionality added	15 (60%)
Move Method	1 (4%)
Rename Method	2 (8%)
Extract Super Class	4 (16%)
Extract Class	7 (28%)
Pull Up Method	3 (12%)

Table 4. Distribution of the 25 Commits that Increased CC.

complexity by itself.

5. Discussion

The quantitative results found in this exploratory study support the assertion made by the original study. We could not find that refactoring commits have a positive effect on the Cyclomatic Complexity metric for the project. The reasons presented below match with the ones raised by Soetens and Demeyer.

In the Apache projects, it is common for developers to perform large commits, with a lot of modifications in the same changeset. One of the reasons of this practice may be related to the use of SVN as their Version Control System (despite Apache provides access to their code base through Git, developers use SVN, Git is just a read-only mirror). This practice may have hidden the real effects of the refactorings performed by developers.

We also noticed that a few developers made bad use of the term *refactoring*. We found out some commits in which the developer affirmed that they were refactorings when, in fact, they were bug fixes or implementation of new features. It can be related to Murphy-Hill et al. [10] work, which affirm that programmers frequently apply what they call *"floss-refactoring"*, which is a refactoring mixed up with other code changes.

Although the quantitative analysis indicates that refactoring does not improve code complexity, when looking to the qualitative analysis, it is possible to say that refactoring actually improve code quality (since complexity is only one code quality attribute). In terms of readability and maintainability, most refactorings had positive effects. There was positive effects even in coupling and cohesion, as many "extract classes" refactorings were found

To sum up, it is no possible to affirm that refactoring does not improve code complexity. There is still work to be done in order to fully understand the behavior of a developer when refactoring, and its general effects in the overall system complexity and code quality.

6. Threats to Validity

There are a few points that needs to be discussed in terms of the validity of this study.

- The proposed heuristic to identify commits that contain refactorings may not identify all of them. Also, it can tell that a commit contains a refactoring when, in fact, it does not. However, Biegel et al. [2] compared different methods of finding code refactorings by using three different similarity measures (text-based, AST-based, and token-based). They conclude that the three techniques achieve similar results.
- During the qualitative analysis, we noticed a few commits that contained a message like *"refactoring to fix ..."*. The developer used the term "refactoring" wrongly. We need to improve the heuristic in order to remove the bad uses of the term.
- The selected sample in the qualitative analysis was small, and may not represent all population of commits. We need to work with a bigger sample, in order to have more confidence in the analysis.
- Sometimes, when refactoring, developers choose to extract code to smaller methods, or even to new and more cohesive classes. When doing it, the number of Cyclomatic Complexity can increase, even though the internal quality improves. It is necessary to make use of different code metrics and see the effects of a refactoring in all of them.
- When calculating the Cyclomatic Complexity of all the projects, we did not ignore test classes. It means that when a commit adds a test class, the CC increases. The algorithm should ignore test classes.
- We analyzed only projects that belong to the Apache Software Foundation. It is necessary to execute the same study in different projects of different domains. It will probably bring new data, and we would be able to do a more complete analysis.

7. Conclusion and Future Work

With the support of MetricMiner, we were able to replicate and extend the original study successfully. In our quantitative study, we found out that Cyclomatic Complexity does not decrease when a developer claims to do a refactoring most of the time – approximatly 23% of the refactorings actually decreased this metric. This finding is similar to the work published by Soetens and Demeyer [11]. On the other hand, when looking to the qualitative data, it is possible to affirm that a refactoring improves the system code in terms of readability and maintainability.

One can argue that Cyclomatic Complexity is not a metric designed specifically to the object oriented world, measuring the complexity of a class as the sum of the CC of its methods might be not completely appropriate. Therefore, common refactorings such as *Extract Method* do not decrease complexity unless it involves removal of code duplication. Nevertheless, extracting a private method from a more complex method might improve the complexity of a class, since it improves its readability.

A future work would be to improve the quantitative analysis and detect refactoring that actually decreased complexity in a more accurate way, mixing different metrics together. Since coupling and methods per class can also be considered complexity metrics in object-oriented systems, metrics measuring these attributes could also be considered in further studies. We could also analyze the evolution of Cyclomatic Complexity in other dimesions such as average complexity per method or per class.

We also observed different patterns of evolution of Cyclomatic Complexity of software projects from the one presented by PMD project. We still need to investigate those patterns in further studies, by modeling the collected data with linear and nonlinear regression models.

Acknowledgment

Marco Gerosa receives individual grant from CNPq.

References

[1] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, Does god class decomposition affect comprehensibility? Proceedings of the IASTED International Conference on Software Engineering, 2006.

[2] Biegel, B.; Soetens, Q. D.; Hornig, W.; Diehl, S.;Demeyer, S. Comparison of similarity metrics for refactoring detection. MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories

[3] Geppert, B.; Mockus, A.; Rossler, F. Refactoring for changeability: A way to go? METRICS 05: Proceedings of the 11th IEEE International Software Metrics Symposium, 2005.

[4] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring, in ESEM 2007: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, 2007.

[5] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[6] Kerievsky, J. Refactoring to Patterns. Addison Wesley, 2004.

[7] Lehman, Meir M.; Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 1980.

[8] M. Alshayeb, Empirical investigation of refactoring effect on software quality. Inf. Softw. Technol., 2009.

[9] Murphy-Hill, E.; Parnin, C.; Black, A. P How we refactor, and how we know it. Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 2009.

[10] Murphy-Hill, E. , Parnin C., and Black, A. P., How we refactor, and how we know it. in ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 2009.

[11] Soetens, Q. D.; Demeyer, S. Studying the Effect of Refactorings: a Complexity Metrics Perspective. The 7th International Conference on Quality in Information and Communications Technology, 2010.

[12] Stroggylos, K.; Spinellis, D. Refactoring - does it improve software quality? WoSQ 07: Proceedings of the 5th International Workshop on Software Quality, 2007.

[13] T. J. McCabe. A complexity measure. IEEE Trans. Softw. Eng., 1976.