What Can Commit Metadata Tell Us About Design Degradation?

Gustavo Ansaldi Oliva
University of São Paulo
Rua do Matão, 1010
São Paulo – SP – Brazil
+55 11 3091-6499
goliva@ime.usp.br

Igor Steinmacher
Federal University of Tech.
Av. Sete de Setembro, 3165
Curitiba – PR – Brazil
+55 44 3523 4156
igorfs@utfpr.edu.br

Igor Wiese
Federal University of Tech.
Av. Sete de Setembro, 3165
Curitiba – PR – Brazil
+55 44 3523 4156
igorfs@utfpr.edu.br

Marco Aurélio Gerosa
University of São Paulo
Rua do Matão, 1010
São Paulo – SP – Brazil
+55 11 3091-6499
gerosa@ime.usp.br

ABSTRACT

Design degradation has long been assessed by means of structural analyses applied on successive versions of a software system. More recently, repository mining techniques have been developed in order to uncover rich historical information of software projects. In this paper, we leverage such information and propose an approach to assess design degradation that is programming language agnostic and relies almost exclusively on commit metadata. Our approach currently focuses on the assessment of two particular design smells: rigidity and fragility. Rigidity refer to designs that are difficult to change due to ripple effects and fragility refer to designs that tend to break in different areas every time a change is performed. We conducted an evaluation of our approach in the project Apache Maven 1 and the results indicated that our approach is feasible and that the project suffered from increasing fragility.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *reestructuring, reverse engineering, and reengineering; version control*.

General Terms

Measurement, Design, Experimentation.

Keywords

Design degradation, software metrics, commit metadata, version control systems, mining software repositories.

1. INTRODUCTION

The volatility of requirements and the wish for new functionalities impose constant pressure for changes in software systems. The need for end-user satisfaction and the great amount of investment leave no room for the software to be largely reworked each time a requirement changes. Indeed, software that is not tolerant to modifications is doomed to abandonment or replacement [3].

Although it is clear that software systems need to be able to evolve, such ability tends to decrease over time [5, 20]. Despite the variety of factors that dictate software evolvability, *software design* has always been in the spotlight. In particular, a great effort has been devoted to the problem of *software design degradation* (a.k.a. design erosion) [12, 16, 20, 11]. Degradation occurs as the elegancy of the system modules becomes lost in the form of structural patches and violations of architectural rules [20]. In this scenario, a dense network of interdependencies among modules commonly emerges, resulting in code that is difficult to change, not reusable, and that does not communicate its intention [7]. As a result, software evolves increasingly slower and its maintenance cost turns to account for a large portion of the total production costs [5, 24].

Researchers and practitioners have long employed static analysis techniques to assess design degradation [1, 4, 10, 14, 15, 18, 27]. Such techniques often rely on an evaluation of successive versions (e.g., revisions, snapshots, or releases) of a software system over time. While these techniques have acknowledged benefits, they also have some drawbacks. Firstly, the input to the static analysis techniques is the source code. Consequently, part of the process necessarily includes parsing such code, which implies that the analysis is not programming language agnostic, i.e., it depends on the specific language in which the subject system was written. This can be a problem itself, since analyzing codebases written in different languages may require different tools and thus lead to more complex analysis processes. This should be true for a company that needs to maintain different projects written in different languages. Another example includes software projects that have components written in different languages, such as a service-oriented system in which components are written in different languages and exposed as web services.

More recently, software repository mining researchers have developed methods, techniques, and tools to uncover rich historical information stored in software repositories, such as version control systems (VCSs), bug-tracking systems, and communication archives (e.g., discussion forums and mailing lists) [13]. In order to complement existing static techniques and deal with their drawbacks, we conceived an approach to asses design degradation that relies almost exclusively on *commit metadata*. Some advantages of our approach include: (i) it is programming language agnostic; (ii) its computation is lightweight; and (iii) its results are based on the actual history of the project. The only requirement is the existence of a VCS (e.g., CVS, Subversion, Git, Mercurial) with enough system development history.

As a first endeavor, our approach focuses on the assessment of two particular design degradation smells introduced by Martin

IWPSE'13, August 19-20, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2311-6/13/08
http://dx.doi.org/10.1145/2501543.2501547

[20]: *rigidity* and *fragility*. Rigidity refers to the tendency for changes in a module to cause a cascade of subsequent changes in the dependent modules. In turn, fragility refers to the tendency for software to break in many different places every time a single change is performed. The reason our approach relies on commit metadata is because the definitions of rigidity and fragility are intrinsically connected to the notion of *change results*. More specifically, in our approach we define metrics that operate on commit metadata to measure rigidity and fragility. Furthermore, since the notion of *degradation* embeds a temporal aspect, we calculate such metrics for every commit included in the development period and analyze their statistical distribution. If we recognize an increase in the values of such metrics, we then infer that design rigidity and fragility are also increasing (i.e., the design is degrading). Therefore, our hypothesis is that information mined from commit metadata is able to reveal design degradation.

In this paper, we introduce our approach and evaluate it by means of an exploratory study. The exploratory study concerned applying the approach to the core of Apache Maven 1.x, whose design degraded so intensively that the development team decided to perform a complete rewrite of the project, which led to Apache Maven 2.0. The goal of such study were twofold: (i) assess the overall feasibility of our approach, and (ii) discover whether our approach is able to show symptoms of increasing rigidity and fragility in Maven 1.x.

The remainder of this paper is organized as follows. In Section II, we introduce the concept of design degradation and present our set of metrics. In Section III, we present the research method we conducted in our study. In Section IV, we present the results of such study. In Section V, we discuss the results we obtained. In Section VI, we present the threats to the validity of our study. In Section VII, we discuss related work. Finally, in Section VIII we state our conclusions and plans for future work.

2. DESIGN DEGRADATION

Martin argued that if dependencies among modules are not adequately managed, then the software starts to rot, just like "a piece of bad meat" [20]. In this scenario, code becomes difficult to be maintained and controlled, the reuse rate and testability decrease, and the effort required to implement new features continually increases. When a design starts to reveal such symptoms, it is said to be degrading or, more informally, rotting. Although similar to the code smells introduced by Fowler [8], these smells are defined at a higher level of abstraction and they impregnate all or great part or the software structure (instead of a localized piece of code).

Martin coined some terms to denote design smells that often appear throughout software development: rigidity, fragility, immobility, and viscosity. In the following subsections, we describe the two particular design smells we are going to deal with in this paper: rigidity and fragility. For each of the concepts, we show the metric we conceived to operationalize it and the rationale behind such operationalization.

2.1 Rigidity

The definition of rigidity given by Martin is as follows. *Rigidity is the tendency for software to be difficult to change, even in simple ways. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules that must be changed, the more rigid the design is. Most developers have faced this situation in one way or another."* [20].

Operationalization of Concept. Software design presents the symptom of rigidity when a single change requires a cascade of subsequent changes in dependent modules. In other words, the more modules one needs to change, the more rigid the design is. Therefore, it becomes natural to measure the intensity of rigidity by calculating the number of modified modules per atomic software change. We operationalize that by calculating the number of changed files per commit, and we refer to this measure as *commit density*.

2.2 Fragility

The definition of fragility given by Martin is as follows. *Fragility is the tendency of a program to break in many different places when a single change is made. Often, the new problems are in areas that have no conceptual relationship with the area that was changed. Fixing those problems leads to even more problems, and the development team begins to resemble a dog chasing its tail* [20].

Operationalization of Concept. Software design presents the symptom of fragility when the software starts to break in many different places every time a single change is performed. In other words, measuring fragility implies identifying and reasoning about the places where changes took place. Therefore, we operationalize fragility by calculating the distance (in the directory tree) among file paths included in a commit. We refer to this measure as *commit dispersion*.

Consider the example depicted in Figure 1. In this figure, shaded nodes denote all existing directories, leaf nodes denote all source code files in the repository tree, and nodes X, Y and Z denote the files that were changed in a specific commit. To calculate commit dispersion, we first calculate the distance between each pair of files in the commit. For example, the distance from Y to Z is 6: 2 steps to go from Y to A, plus 4 steps to go from A to Z. Commit dispersion is then given by the average of the distances, i.e., the sum of pair-wise distances (X to Y, X to Z, and Y to Z) divided by the number of pairs (3). The result for this example is (3 + 7 + 6) / 3 = 5.333.

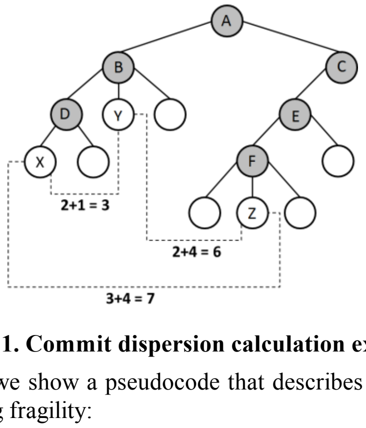

Figure 1. Commit dispersion calculation example

In Listing 1, we show a pseudocode that describes the algorithm for calculating fragility:

```
Routine: calculateCommitDispersion
Parameters: commit
01. totalDistance ← 0
02. commitDispersion ← 0
03. filePairs ← 0
04. numFiles ← commit.getNumFiles()
```

```
05. for (i = 0; i < numFiles; i++)
06.   for (j = i + 1; j < numFiles; j++)
07.     //Obtains the leaf nodes
08.     lfNodeA ← commit.getFile(i)
09.     lfNodeB ← commit.getFile(j)
10.     //Obtains the Lowest Common Ancestor(LCA)
11.     lca ← obtainLCA(lfNodeA, lfNodeB
12.     //Calculates distance between leaf nodes
13.     distance ← calcDist(lfNodeA,lfNodeB,lca)
14.     totalDistance ← totalDistance + distance
15.     filePairs ← filePairs + 1
16.   end-for
17. end-for
18. //Calcs avgDistance between nodes in commit
19. if (filePairs > 0)
20.   commitDispersion ← totalDistance/filePairs
21. end-if
22. return commitDispersion
```

Listing 1. Commit Dispersion Algorithm

The distance between two leaf nodes (line 13) is calculated by summing the distance of each node to their common lowest ancestor node (LCA) in the tree. Clearly, the LCA is always a directory node.

3. RESEARCH METHOD

In this section, we describe the exploratory study we conducted. We first searched for a software system whose design presented acknowledged symptoms of rigidity and fragility. After having found such system, we mined its version control system using XFlow [25] and then calculated the metrics we conceived for measuring design degradation. Our goal was to investigate the feasibility of our approach and check whether we would indeed find an increase in the metric values throughout the considered development period of the system.

3.1 The Subject System: Apache Maven 1

For this study, we needed a software project that satisfied all of the following requirements: (i) to present evident symptoms of design degradation; (ii) to be hosted on a Subversion (SVN) repository with anonymous read access; (iii) non-academic; (iv) to have a non-small development history. The first requirement arises from the very nature of the goal of our study. The second requirement exists due to practical constraints on the tools at our disposal. The third requirement was raised because we wanted to evaluate the metrics on a real-world software project, which could be either an industrial software or a free/libre open source software (FLOSS). The fourth and last requirement exists because we want to conduct our evaluation on a software system with enough development history.

The project we selected was (the core of) Apache Maven 1.x. Maven is a tool aimed at supporting project management and providing build automation. Although Maven functionalities are roughly similar to those of the software Ant, it is based on different concepts. In particular, Maven relies on the concept of Project Object Model (POM), and thus is able to manage a project's build, reporting, and documentation from a centralized piece of information. Maven is currently an Apache Software Foundation (ASF) top-level project.

Regarding the first requirement, Apache Maven 1.x evolved until a complete rewrite was needed due to signs of design degradation. Indeed, in the Maven 1.x project website, the authors wrote that "the latest version of Maven is the 2.0 tree, which is a complete rewrite of the original Maven application." In the release notes of Maven 2.0, there is also a similar statement made by the authors "Maven 2.0 is a rewrite of the popular Maven application to achieve a number of new goals, and to provide a stable basis for future development." We surveyed few Maven developers by email and the answers received reveal that Maven 1.x was monolithic and hard to maintain. A developer said it was "unmaintainable" and another one highlighted that "build scripts were getting too complex". They also reported that rigidity and fragility highly influenced Maven redesign.

Regarding the second requirement, Apache Maven 1.x was hosted in a Subversion code repository. Regarding the third requirement, Maven was (and has been) maintained by the Apache Software Foundation (ASF), which is a non-profit organization that has developed nearly a hundred distinguishing software projects that cover a wide range of technologies and address several problems from diverse contexts. With relation to the fourth and last requirement, we used XFlow to mine history information from the project and we discovered that it involved 37 developers who performed 6,753 commits in total. The development history encompassed approximately 6 years (from February 2002 to February 2008). In particular, according to the categorization proposed by Levine and Moreland [17], small teams are groups of 5 to 15 individuals. Hence, in light of such categorization, the selected project fully satisfies the pre-established criterion of development team size.

3.2 Main Steps

In this subsection, we describe the main steps we followed in our study.

Data collection and pre-filtering. Interacting with remote version control systems is usually both troublesome and slow. Furthermore, to make things even more complicated, ASF has a single SVN repository that hosts all of its projects and comprises more than 1.3 million commits. Given these two aspects, we decided to mirror the whole ASF SVN repository and then interact with it locally. After having finished the time-consuming mirroring process, we used XFlow to parse Maven 1 commits and store its metadata into a MySQL database. Since we are interested in evaluating design degradation, we only considered java files when doing the commit parsing (i.e., all other kinds of files were discarded). Commits having no java files were simply discarded. Furthermore, we were interested in all commits done in branches, tags, and trunks. Therefore, we considered java files inside the root folder "/maven/maven-1/core". As a final remark, we highlight that all these filters were easily applied on-the-fly during data collection thanks to the flexibility provided by XFlow [25].

Post-filtering. The approach we conceived to calculate rigidity and fragility operates directly on commit metadata. Consequently, we needed to exclude commits that referred crosscutting changes, such as applying or changing software license, doing repository merge operations, and fixing code styling issues. Hence, after data collection, we applied some heuristics to exclude these non-wanted commits. For this particular project, we discarded commits that included the words "cvs2svn', "ASL", "license header", and "m2 code style" in their comments. We also discarded those commits whose comments started with "Initial revision". We came up with this set of keywords by applying the following strategy. We first discovered which were the largest commits (in terms of commit density), selected three to five of those, and then manually inspected their metadata (focusing on the list of changed files and the author comments). As a result, we conceived a filtering keyword and tested it to check whether it was able to select the commits we wanted. In case it selected more commits than it should, we inspected the additional commits and adjusted the keyword accordingly. In fact, in some situations the keyword spot more commits than we had initially identified. Once the keyword was deemed ok, we repeated the whole process until only desired commits were left. We emphasize that we were not able to come up with a simple keyword to capture commits that referred to repository merge operations. In such case, the strategy was to discover all commits that included the word "merge" in their comments, inspect their metadata, and decide which ones were actually related to repository merge operations.

Data analysis perspectives. Once we finished the data collection process, we defined which data analysis perspectives we would employ. After reasoning about the project characteristics, we decided to follow a top-down approach by (i) first analyzing the development period as a whole, then (ii) diving the development period into three slots with each containing the same amount of commits, and finally (iii) dividing the development period into three slots with each referring to a major release of the software. Given the Maven release history depicted in Table 1, we defined the three perspectives as follows:

- Whole development period: This period goes from the start of development (19/02/2002) until the end of development (03/02/2008).

Table 1. Apache Maven Release History

Version	Date (dd/mm/yyyy)
[Start of Development]	19/02/2002
Maven 1.0 Beta 5	12/08/2002
Maven 1.0 Beta 6	20/08/2002
Maven 1.0 Beta 7	30/09/2002
Maven 1.0 Beta 8	12/02/2003
Maven 1.0 Beta 9	08/04/2003
Maven 1.0 Beta 10	14/07/2003
Maven 1.0 RC 1	29/09/2003
Maven 1.0 RC 2	23/03/2004
Maven 1.0 RC 3	19/05/2004
Maven 1.0 RC 4	28/06/2004
Maven 1.0	13/07/2004
Maven 1.0.1	10/11/2004
Maven 1.0.2	07/12/2004
Maven 1.1 Beta 1	17/06/2005
Maven 1.1 Beta 2	09/09/2005
Maven 1.1 Beta 3	02/08/2006
Maven 1.1 RC 1	11/05/2007
Maven 1.1	25/06/2007
[End of Development]	03/02/2008

- Three commit groups of equal size: The data collection procedure resulted in 2145 commits. Hence, the first commit group comprised commits 1-715, the second commit group comprised commits 716-1430, and the last commit group comprised commits 1431-2145. In this case, the goal was to divide all the contribution volume chronologically into three equal-sized blocks and check whether the metrics values would change from block to block.

- Three distinct development periods: The first development period goes from the start of development (19/02/2002) until the release of Maven 1.0 Beta 5 (12/08/2002). The second development period goes from the day after the release of Maven 1.0 Beta 5 (20/02/2002) until the release of Maven 1.0.2 (07/12/2004). The last development period goes from the day after the release of Maven 1.0.2 (08/12/2004) until the end of development (03/02/2008). In this case, we the goal was to check whether metric values would change from release to release.

Data analysis procedures. We organized data according to each of the aforementioned analysis perspectives and imported it into MS Excel and Minitab. From these tools, we calculated descriptive statistics, plotted graphs, and discovered trends. In particular, we tried to understand how our measures of rigidity and fragility changed over time, i.e., we studied their statistical distribution.

3.3 Supporting Tools

XFlow. Mining repositories studies usually require extensive tool support due to large and complex data that need to be collected, processed, and analyzed [2]. XFlow is an extensible and interactive open source tool whose general goal is to provide a comprehensive analysis of software projects evolution process by mining software repositories and taking into account both technical and social aspects of the developed systems [25].

Rigidity and Fragility Calculator. We implemented a Java standalone prototype tool to calculate rigidity and fragility for every commit of the subject system.

Microsoft Excel, Minitab, and R. These tools supported all statistical analyses we performed. The statistical tests were performed in Minitab (except for the *Augmented Dickey–Fuller test* and the *Powerlaw test*, which were available only in R).

4. RESULTS

In this section, we show the results we obtained by applying our approach to the core of the Apache Maven 1 project.

4.1 Rigidity

4.1.1 Whole development period

Table 2 depicts the descriptive statistics for commit density when considering the whole development period.

Table 2. Descriptive Statistics for Commit Density

N	Mean	StDev	Min	Q1	Med.	Q3	Max	Skew	Kurt.
2145	2.60	5.46	1	1	1	2	79	7.81	78.42

Given the values of Q1 and Q3, it follows that the lower and upper whiskers of the data corresponding boxplot are 1 and 3 respectively. In other words, usual commit density values range from 1 to 3. Indeed, in terms of cumulative percentage, commits with 1, 2, and 3 files comprise 86% of all data. This can be easily seen in the frequency histogram for commit density depicted in Figure 2.

We tried to identify the data distribution by performing goodness of fit tests against the following known distributions: *Normal*, *Normal after Box-Cox Transformation*, *Lognormal*, *3-Parameter Lognormal*, *Exponential*, *2-Parameter Exponential*, *Weibull*, *3-Parameter Weibull*, *Smallest Extreme Value*, *Largest Extreme Value*, *Gamma, 3-Parameter Gamma, Logistic*, *Loglogistic*, *3-*

Parameter Loglogistic, and *Powerlaw*. The only good fit we found was for the PowerLaw distribution (as somehow suggested by the histogram depicted in Figure 2). In all other cases, we could always reject the null-hypothesis with high confidence.

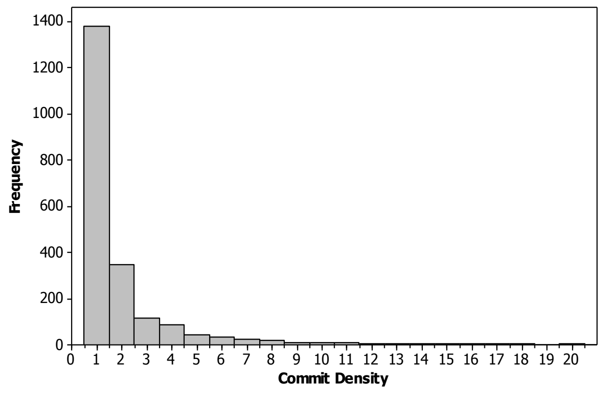

Figure 2. Histogram for Commit Density

Figure 3 shows a time-series plot for commit density. Analyzing the distribution, we noticed that 80.7% values are below `mean` (first threshold), 95.6% are below `mean + 1*StdDev` (second threshold), and that 97.4% of the values are below `mean + 2*StdDev` (third threshold). It is interesting to notice, however, that most part of the values above the third threshold occur in the second-half portion of the commits (i.e., from commit 1073 onwards).

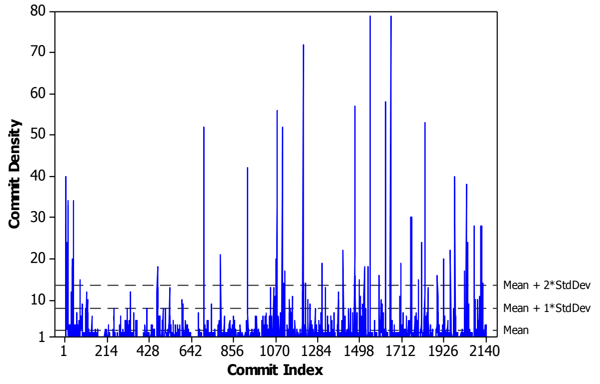

Figure 3. Time Series Plot for Commit Density

We performed a trend analysis for commit density and tried four different regression models: *linear*, *quadratic*, *exponential growth*, and *s-curve* (Pearl-Reed logistic). For each model, we computed three accuracy measures: *Mean Absolute Percentage Error* (MAPE), *Mean Absolute Deviation* (MAD), and *Mean Squared Deviation* (MSD). The results we obtained were as follows:

Table 3. Trend Analysis for Rigidity

#Trend Model	MAPE	MAD	MSD
Linear	116.22%	2.24	29.74
Quadratic	115.99%	2.24	29.69
Exponential	53.47%	1.75	30.80
S-Curve	39.61%	1.68	31.46

By analyzing the results, we notice that the S-curve model had the best fit (even though it showed the worst MSD). In this curve, the first commit scored 1.14 and the last one scored 1.33, which indicates a negligible trend of increase for the commit density metric.

Finally, we ran the *Augmented Dickey-Fuller* test, which tests for a unit root in a time series sample. The results are summarized below and indicate that we can reject the null hypothesis, which states that the time series has a unit root. In other words, according to the test, the sample can be deemed trend-stationary.

```
Augmented Dickey-Fuller Test
data:  rigidity
Dickey-Fuller  =  -12.3017,  Lag  order  =  12,
p-value = 0.01
alternative hypothesis: stationary
```

4.1.2 Three commit groups of equal size

Table 4 depicts the descriptive statistics for commit density when considering three commit groups of equal size.

Table 4. Descriptive Statistics for Commit Density (three groups of equal size)

#ID	N	Sum	Sum%	Mean	StDev	Min	Max	Skew	Kurt.
G1	715	1643	29.5%	2.30	4.13	1	52	6.70	56.71
G2	715	1757	31.6%	2.46	4.92	1	72	8.77	97.11
G3	715	2166	38.9%	3.03	6.92	1	79	6.84	57.68

An analysis of the data shows that the sum of the number of files per commit increases from group to group. Consequently, there is a minor increase in the mean values (2.30, 2.46, 3.03). Furthermore, the maximum value also increases (52, 72, 79). However, from the perspective of rigidity analysis, these results do not mean much given the high standard deviation values.

We performed a quartile analysis (Table 5) and we noticed that the quartile values and upper whisker were identical for the three groups. On the other hand, we noticed an increase in the number of outliers from group to group (we say that a value is an outlier when it exceeds the upper whisker). Furthermore, the sum of the values of the outliers also increased. Therefore, although we did not find any striking evidence for the increase of rigidity in the previous analyses, this last result shows that at least the occurrence of large commits increased over time.

Table 5. Quartile Analysis for Commit Density (three groups of equal size)

#ID	N	Q1	Med.	Q3	Upper Whisker	#Outliers	%Outliers	Sum of Outliers
G1	715	1	1	2	3	96	13.4%	858
G2	715	1	1	2	3	99	13.8%	925
G3	715	1	1	2	3	105	14.7%	1358

Finally, Figure 4 depicts a time-series plot for commit density that highlights the division of commit into groups.

4.1.3 Three distinct development periods

Table 6 depicts the descriptive statistics for commit density when considering the release periods.

Table 6. Descriptive Statistics for Commit Density (per release period)

#ID	N	N%	Sum	Sum%	Mean	StDev	Min	Max	Skew	Kurt.
G1	888	41.4%	1952	35.1%	2.20	3.81	1	52	7.08	64.67
G2	1132	52.8%	3218	57.8%	2.84	6.40	1	79	7.58	69.63
G3	125	5.8%	396	7.1%	3.17	5.97	1	38	3.91	15.95

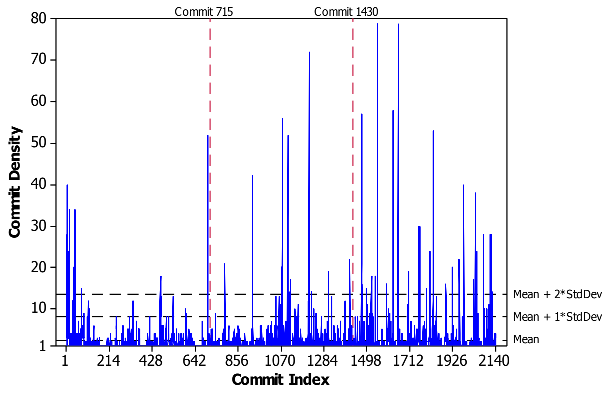

Figure 4. Time Series Plot for Commit Density (three groups of equal size)

An analysis of the data shows that the third group is considerably smaller than the other two (in terms of the number of commits). Despite this difference, we again notice a minor increase in the mean values (2.20, 2.84, 3.17) from group to group. As in the previous section, these results do not mean much given the high standard deviation values.

We performed a quartile analysis and the results are summarized in Table 7. As in the previous section, quartile values and upper whisker were identical for the three groups. On the other hand, despite the different sizes of the groups, we noticed a percentage increase of outliers from group to group. Therefore, although we did not find any striking evidence for the increase of rigidity in the previous analyses, this last result again shows that at least the occurrence of large commits increased (proportionally) over time.

Table 7. Outlier Analysis (per release period)

#ID	N	Q1	Med.	Q3	Upper Whisker	#Outliers	%Outliers	Sum of Outliers
G1	888	1	1	2	3	114	12.8%	964
G2	1132	1	1	2	3	167	14.8%	1921
G3	125	1	1	2	3	19	15.2%	256

Finally, Figure 5 depicts a time-series plot for commit density that highlights the division of commits according to the release periods we previously defined.

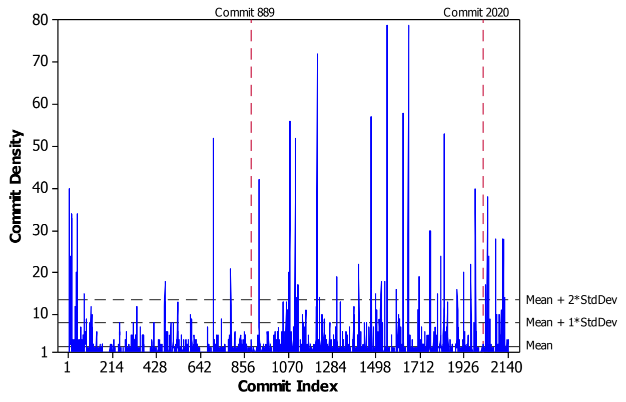

Figure 5. Time Series Plot for Commit Density (per release period)

4.2 Fragility

4.2.1 Whole development period

Table 8 depicts the descriptive statistics for commit dispersion when considering the whole development period.

Table 8. Descriptive Statistics for Commit Dispersion

N	Mean	StDev	Min	Q1	Med.	Q3	Max	Skew	Kurt.
2145	2.11	4.44	0	0	0	2	21	2.63	6.51

Given the values of Q1 and Q3, it follows that the lower and upper whiskers of the data corresponding boxplot are 0 and 5 respectively. In other words, usual commit dispersion values range from 0 to 5. Indeed, in terms of cumulative percentage, commits with dispersion from 0 to 5 comprise 87% of all data. This can be easily seen in the frequency histogram for commit dispersion depicted in Figure 6.

As in the case of rigidity, we tried to identify whether commit dispersion followed any known statistical distribution. In all the tests we performed, we could always reject the null-hypothesis. In other words, commit dispersion did not fit any known standard statistical distribution.

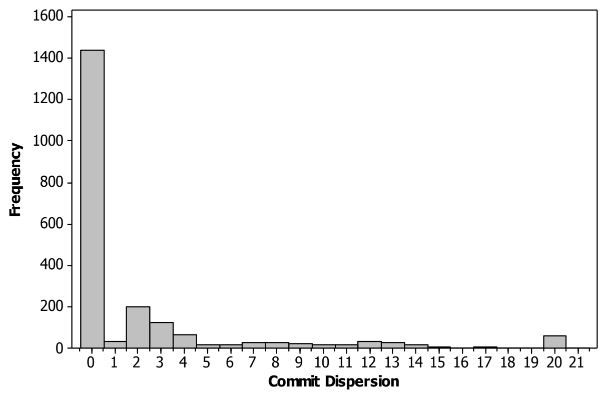

Figure 6. Histogram for Commit Dispersion

Figure 7 shows a time-series plot for commit dispersion. Analyzing the distribution, we concluded that 76.8% of the values are below `mean` (first threshold), 88.0% are below `mean + 1*StdDev` (second threshold), and that 92.6% of the values are below `mean + 2*StdDev` (third threshold). Interestingly, commit dispersion becomes much higher from commit 1233 onwards.

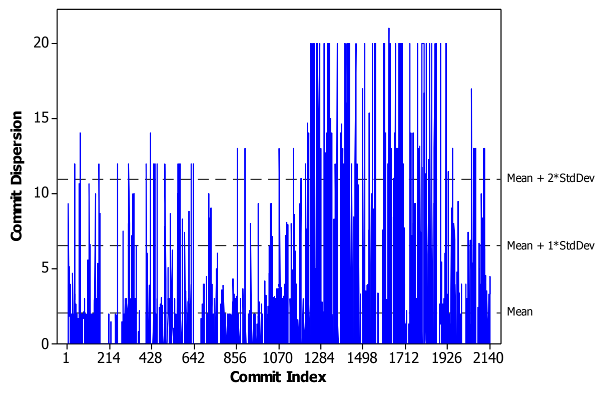

Figure 7. Time Series Plot for Commit Dispersion

We performed a trend analysis for commit dispersion and tried the same regression models from Section 4.1 (except for *exponential growth*, which requires all values to be positive). The results we obtained were as follows:

Table 9. Trend Analysis for Fragility

#Trend Model	MAPE	MAD	MSD
Linear	52.69%	2.87	19.12
Quadratic	53.50%	2.87	19.11
S-Curve	223.89%	9.06	91.95

The linear model had the best fit. In this curve, the first commit scored 2.30 and the last one scored 3.42, which indicates a slight trend of increase for the commit dispersion metric.

As we did for the rigidity metric, we ran the *Augmented Dickey-Fuller* test. The results are summarized below and indicate that we can reject the null hypothesis, which states that the time series has a unit root. In other words, according to the test, the sample can be deemed trend-stationary.

```
Augmented Dickey-Fuller Test
data:  fragility
Dickey-Fuller  =  -9.7536,  Lag  order  =  12,
p-value = 0.01
alternative hypothesis: stationary
```

4.2.2 Three commit groups of equal size

Table 10 depicts the descriptive statistics for commit density when considering three commit groups of equal size.

Table 10. Descriptive Statistics for Commit Dispersion (three groups of equal size)

#ID	N	Sum	Sum%	Mean	StDev	Min	Max	Skew	Kurt.
G1	715	854.6	18.9%	1.20	2.65	0	14	2.90	8.36
G2	715	1546.0	34.1%	2.16	4.53	0	20	2.67	6.67
G3	715	2127.8	47.0%	2.98	5.49	0	21	1.99	2.92

An analysis of the data shows that the sum of the commit dispersion increases substantially from group to group. Consequently, there is an increase in the mean values (1.20, 2.16, 2.98). Furthermore, the maximum value also increases (14, 20, 21). However, from the perspective of fragility analysis, these results should be interpreted with care because of the high standard deviation values.

To further investigate the situation, we performed a quartile analysis. Differently from rigidity, the results shown in Figure 8 indicate that fragility increased from group to group. In particular, the third quartile increased from 1.73 to 2.0 to 3.33 and the upper whisker increased from 4.0 to 5.0 to 8.0.

Table 11 depicts an analysis of outliers. We noticed an increase in the absolute number (and consequently, in the percentage of) outliers from group to group. Furthermore, the sum of the values of the outliers also increased. This provides more evidence that fragility increased over time.

Table 11. Outliers Analysis (three groups of equal size)

#ID	N	Number of Outliers	Percentage of Outliers	Sum of Outliers
G1	715	56	7.8%	518.0
G2	715	87	12.2%	1131.4
G3	715	108	15.1%	1590.0

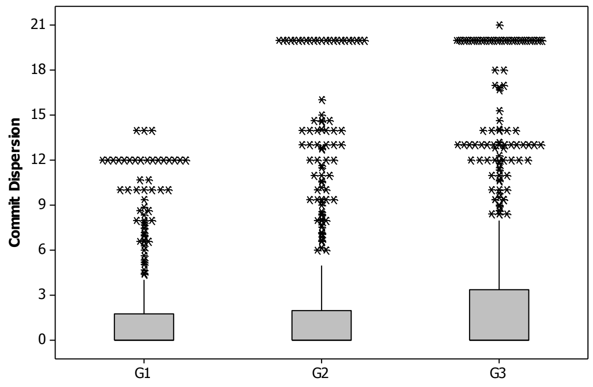

Figure 8. Boxplots for Commit Dispersion (three groups of equal size)

Finally, Figure 9 depicts a time-series plot for commit dispersion that highlights the division of commits into groups. In particular, we analyzed their distribution using Minitab and we concluded that none of the three groups follows any known standard statistical distribution (including common transformations).

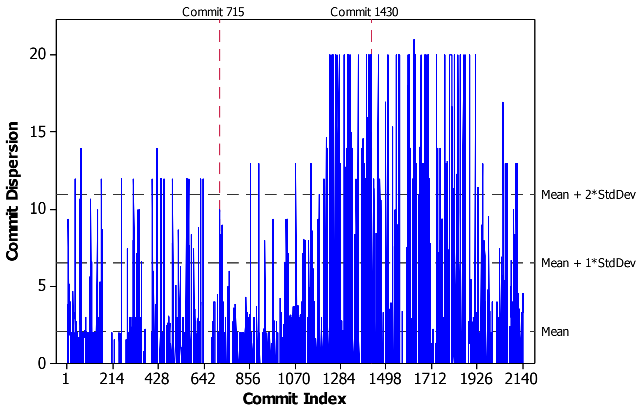

Figure 9. Time Series Plot for Commit Dispersion (three groups of equal size)

4.2.3 Three distinct development periods

Table 12 depicts the descriptive statistics for commit dispersion when considering the release periods.

Table 12. Descriptive Statistics for Commit Dispersion (three groups of equal size)

#ID	N	N%	Sum	Sum%	Mean	StDev	Min	Max	Skew	Kurt.
G1	888	41.4%	1006.6	22.2%	1.13	2.52	1	14	3.02	9.39
G2	1132	52.8%	3268.6	72.2%	2.89	5.45	1	21	2.07	3.21
G3	125	5.8%	253.2	5.6%	2.03	3.49	1	17	2.14	4.61

An analysis of the data shows that the third group is considerably smaller than the other two (in terms of the number of commits). The mean value increases from the first group to the second, and then decreases from the second to the third.

We performed a quartile analysis and the results shown in Figure 10 indicate that fragility increased from group to group. In particular, the third quartile increased from 1.71 to 3.0 to 3.33 and the upper whisker increased from 4.0 to 7.5 and then stayed the same for the third group.

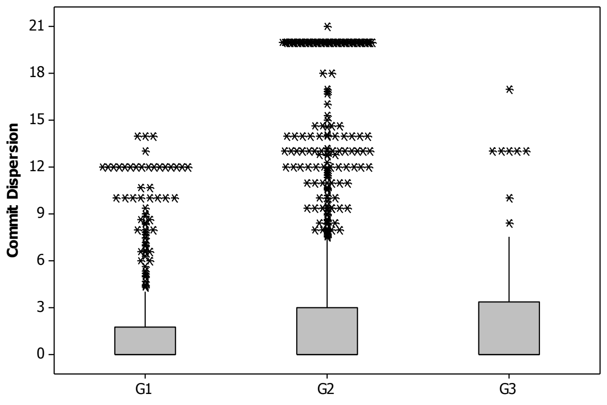

Figure 10. Boxplots for Commit Dispersion (per release period)

Table 13 depicts an analysis of outliers. We noticed an increase in the absolute number and in the percentage of outliers from the first group to the second group. Furthermore, the sum of the values of the outliers also increased from the first group to the second. The third group had only 125 outliers, which corresponds to 6.4% of all observations in this group. This provides more evidence that fragility increased over time and that it was more evident in the second release of the project.

Table 13. Outlier Analysis (per release period)

#ID	N	Number of Outliers	Percentage of Outliers	Sum of Outliers
G1	888	63	7.1%	573.7
G2	1132	178	15.7%	2556.0
G3	125	8	6.4%	100.4

Finally, Figure 11 depicts a time-series plot for commit dispersion that highlights the division of commits into the groups corresponding to the releases. In particular, we analyzed their distribution using Minitab and we concluded that none of the three groups follows any known standard statistical distribution (including common transformations).

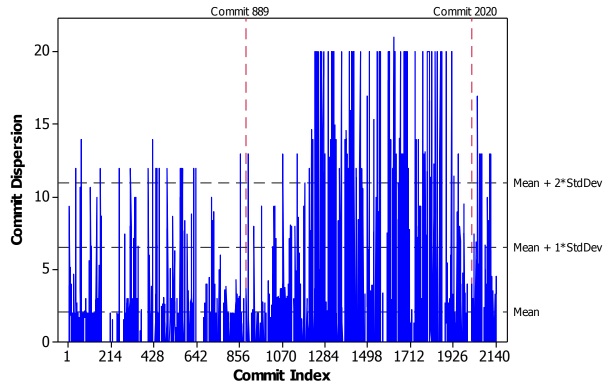

Figure 11. Time Series Plot for Commit Dispersion (per release period)

5. DISCUSSION

In this paper, we intended to (i) assess the overall feasibility of our approach, and (ii) discover whether our approach is able to show symptoms of increasing rigidity and fragility in Maven 1.x. Regarding (i), we noticed that our approach was feasible, since we were able to quickly calculate the metrics and obtain the raw results. At the same time, we had to face some issues. Firstly, we realized that the data needs to be pre-processed and post-processed (Section 3.2), otherwise the metrics we proposed get influenced by dirty data and do not provide reliable results. Automating this task seems often hard, since the filtering process operates on contextual information. In the cases where commit policies are enforced, this task should become easier. Secondly, we noticed that an in-depth interpretation of the results require some statistical background on data distributions and forecasting. On the other hand, the rationale behind such interpretation could be encapsulated into the approach itself (i.e., it could be implemented), so that the output is given to the end-user in a more friendly and straightforward way. Another option would be to rely on reference values calculated over projects that share a similar context. Once either option is chosen, it should not be too complicated to embed such analysis into a continuous integration process. Regarding (ii), we applied the approach to Maven 1.x and obtained the results summarized in Table 14. P1 stands for the first perspective analysis (whole project), P2 stands for the second one (three commits groups of equal size), and P3 stands for the third one (release periods). The word *increase* in the table header means that there was an increase in the value from G1 to G2, and also from G2 to G3.

Table 14. Summary of Findings

	Trend of Increase	Increase in Med.		Increase in IQR		Increase in the % of Outliers		Increase in Mean	
	P1	P2	P3	P2	P3	P2	P3	P2	P3
Rigidity						✓	✓	✓	✓
Fragility	✓			✓	✓	✓		✓	

We did not find many evidences of rigidity in the project. We only found an increase in the percentage of outliers and in the mean values, which occurred for both P2 and P3. It caught our attention however that this happened even though the third release comprises a very limited number of commits (5.8%). In the case of fragility, the results were stronger. We found a slight trend of increase using a linear regression model and we also found an increase in the interquartile range (for both P2 and P3). We believe we did not find an increase in the median because more than 60% of the commits include only a single file, which results in zero fragility according to our metric. Given the time series plot of fragility, we already expected to see an increase in the percentage of outliers and in the mean for P2. Indeed, this time series plot reveal that fragility increased a lot during the second release (G2) and started to decrease in the third release (G3).

6. THREATS TO VALIDITY

Some factors may have influenced the validity of our study. In the following, we present such factors.

Construct validity. Since the concepts of rigidity and fragility are broad and complex, it can be that our operationalization do not accurately represent reality. Furthermore, mismatches can occur depending on how effectively commit policies are enforced. In particular, for projects in which different developers have different commit habits, our metrics may derive misleading results (just consider one developer that commits his changes very frequently and another one that only commits his changes by the end of his working day). To mitigate this issue, automated commit grouping strategies could be employed to merge related commits [21].

Internal validity. There is a threat related to the way we analyzed the project, since different configurations of commit groupings could possibly generate different results. The pre and post filtering processes directly influence the input to the approach, which means that other filtering keywords could possibly change the results we obtained. Furthermore, it could be that unknown contextual factors actually caused our metric values to change over time, thus disconnecting them from the purpose of detecting symptoms of degrading design.

Conclusion validity. Despite the acknowledged degradation of Maven's 1 design, we found no strong evidence of increasing rigidity throughout the project's history. In fact, it can be that the design did not actually suffer from rigidity at all. We sent a questionnaire to the developers listed in the "Maven Team", including the responsible for the redesign, to hear their opinion on the reasons the design degraded. We received answers from only four developers. They confirmed the design degradation, but they informed that they did not directly contribute to the redesign process. Unfortunately, we were not able to get the feedback from the key developer of Maven 1.x that was the actual main responsible for the redesign.

External validity. The findings of this study are limited to the evaluation of a single software system, thus constraining the external validity of this study. In fact, the goal of this study was to perform a first assessment of the feasibility and effectiveness of the approach.

7. RELATED WORK

The phenomenon of design degradation has been noted since the early days of Software Engineering. Along more than twenty years (1974-1992), Lehman and colleagues [16] proposed the laws (or rather empirical hypotheses) of software evolution, being the first ones specifically concerned with continuing change and increasing complexity. In 1992, inspired by the second law of thermodynamics, Jacobson [12] coined the term software entropy to refer to the increases in software disorder (entropy) over time. In 1994, Parnas [23] introduced the idea of *software aging*, by arguing that programs get old, just like people.

The phenomenon of design rigidity has also been studied under the name of ripple effect. An early work on software ripple effect is that of Yau and colleagues [28], who presented a maintenance framework to cope with program modifications. Wilkie and Kitchenham [27] investigated whether classes with high CBO (Coupling Between Objects) metric values are more likely to be affected by change ripple effects. Similarly, Briand and colleagues [4] investigated the use of coupling measures and derived decision models for identifying classes likely to suffer from ripple effect. Interestingly, these two last studies revealed that highly structurally coupled classes did not always cause significant ripple effects. Therefore, we believe that our proposed rigidity metric may complement such existing approaches. For instance, the tool IBM Structural Analysis for Java (SA4J) tool offers an interactive visualization named "What If" that highlights ripple effects based on existing structural dependencies between classes (Figure 12). Conceptual coupling metrics, which are calculated based on semantic information obtained from identifiers and comments in source code, have also been employed to detect ripple effect [14].

Design fragility has also been studied under the name of Shotgun Surgery [8]. We highlight the work of Lanza and colleagues [15, 19] in this area, who proposed mechanisms called "detection strategies" that combine different code metrics to detect code smells. The free iPlasma tool [18], developed by Lanza and the LOOSE Research Group, implements the Shotgun Surgery detection strategy (Figure 13) by parsing the source code of Java and C# projects. Gîrba and colleagues [10] proposed a similar approach to detect this same smell based on the identification of classes that have had their implementation changed together while maintaining their interfaces intact. While all these studies depend on the actual code, our proposed metric relies on commit metadata obtained through the parsing of the log files generated by the version control system. Therefore, the calculation of our metric is fast and does not depend on the programming language in which the software was written.

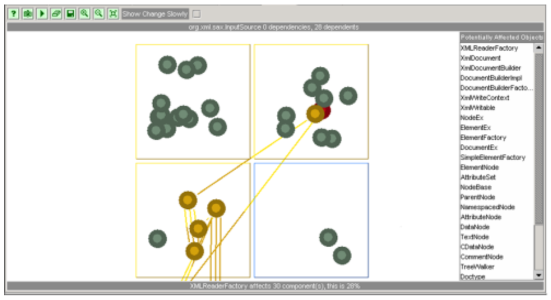

Figure 12. IBM Structural Analysis for Java ("What if" Visualization)

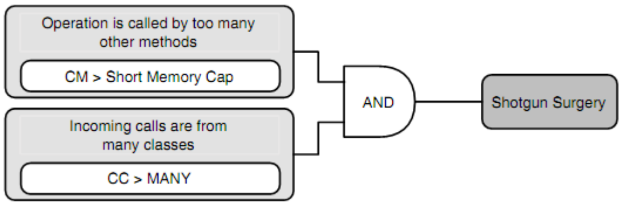
Operation is called by too many other methods
CM > Short Memory Cap
Incoming calls are from many classes
CC > MANY
AND
Shotgun Surgery

Figure 13. Shotgun surgery detection strategy [15, 19]

Finally, in a more general context, Gall and colleagues [9] mined CVS repositories, collected logical dependencies (a.k.a. evolutionary dependencies [22]), and showed that design flaws such as God Classes [8] and Spaghetti Code [7] could be discovered without analyzing the actual source code. D'Ambros and colleagues developed an interactive visualization tool called Evolution Radar [6] that displays logical dependencies among modules of a software system. They showed that their tool was able to detect design issues that were not detectable by means of static analysis of code.

8. CONCLUSIONS AND FUTURE WORK

Mining-based approaches leverage historical data stored in software repositories to uncover rich evolutionary information. In this study, we proposed an approach that relies on a set of metrics that operate on commit metadata to assess design degradation. The results of the evaluation involving the core of the Apache Maven 1.x showed that our approach is feasible and that the project suffered from increasing fragility. Such outcome suggests that mining-based approaches should be further developed and enhanced to complement existing structural analysis techniques, since the two different approaches capture different dimensions of software evolution. As future work, we believe that the approach should be further evaluated and refined based on the analysis of other projects. Gathering feedback from developers using qualitative research methods seems essential for a thorough evaluation of the approach. Furthermore, it would be interesting to compare the metric values for Maven 1.x and Maven 2.x, since the latter intended to avoid known problems from Maven 1.x and provide a "stable basis for future development". Finally, building tools to automate the metrics calculation and results interpretation is essential for real-world adoption. We foresee some research opportunities:

Technical Debt. Since our approach quantifies rigidity and fragility, it could be employed in the calculation of technical debt (design debt) [26].

Design immobility. Our approach could be extended to calculate design immobility, which is another design smell introduced by Martin [20] (and long known by the software development community).

Candidates for refactoring. The number of times each file is committed seem to be a good heuristic for automatically detecting good refactoring candidates. For instance, Figure 14 shows the number of commits per file in the core of Maven 1. The right-hand side of the chart point to highly mutable files that are natural candidates for refactoring, since they were committed much more often as compared to other files of the same system.

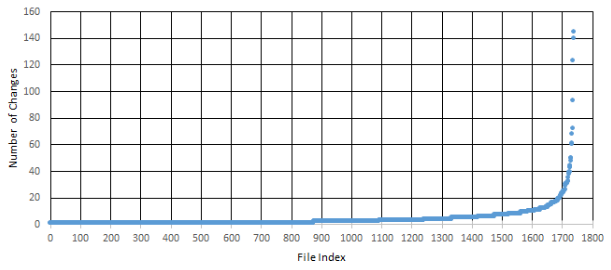

Figure 14. Number of commits per file in Maven

9. ACKNOWLEDGEMENTS

Gustavo Oliva receives individual grant from the CHOReOS EC FP7 project. Marco Gerosa receives individual grant from CNPq. Igor Steinmacher and Igor Wiese thank Fundação Araucária for its Financial Support.

10. REFERENCES

[1] Arnold, R.S. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press.

[2] Bevan, J. et al. 2005. Facilitating software evolution research with kenyon. *SIGSOFT Softw. Eng. Notes*. 30, 5 (Sep. 2005), 177–186.

[3] Booch, G. et al. 2007. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional.

[4] Briand, L.C. et al. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Softw. Eng.* 25, (Jan. 1999), 91–121.

[5] Brooks, F.P. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional.

[6] D'Ambros, M. et al. 2009. Visualizing Co-Change Information with the Evolution Radar. *IEEE Trans. Software Eng*. 35, (2009), 720–735.

[7] Foote, B. and Yoder, J.W. 1999. *Pattern Languages of Program Design*. Addison-Wesley Professional.

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

[9] Gall, H. et al. 2003. CVS Release History Data for Detecting Logical Couplings. *Proceedings of the 6th International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2003), 13–.

[10] Gîrba, T. et al. 2007. Using concept analysis to detect co-change patterns. *9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, 2007* (2007), 83–89.

[11] Gurp, J. van et al. 2005. Design preservation over subsequent releases of a software product: a case study of Baan ERP: Practice Articles. *J. Softw. Maint. Evol.* 17, (Jul. 2005), 277–306.

[12] Jacobson, I. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional.

[13] Kagdi, H. et al. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.* 19, 2 (Mar. 2007), 77–131.

[14] Kagdi, H. et al. 2010. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. *Reverse Engineering (WCRE), 2010 17th Working Conference on* (2010), 119–128.

[15] Lanza, M. and Marinescu, R. 2006. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

[16] Lehman, M. et al. 1997. Metrics and Laws of Software Evolution–The Nineties View. *Proceedings IEEE International Software Metrics Symposium (METRICS'97)* (Los Alamitos CA, 1997), 20–32.

[17] Levine, J.M. and Moreland, R.L. 1990. Progress in Small Group Research. *Annual Review of Psychology*. 41, (1990), 585–634.

[18] Marinescu, C. et al. 2005. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) - Industrial and Tool volume* (2005), 77–80.

[19] Marinescu, R. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2004), 350–359.

[20] Martin, R.C. and Martin, M. 2006. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall.

[21] Oliva, G.A. et al. 2012. Preprocessing Change-Sets to Improve Logical Dependencies Identification. *Proceedings of the 6th International Workshop on Software Quality and Maintainability* (Szeged, Hungary, 2012).

[22] Oliva, G.A. et al. 2011. Towards a classification of logical dependencies origins: a case study. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution* (Szeged, Hungary, 2011), 31–40.

[23] Parnas, D.L. 1994. Software aging. *Proceedings of the 16th international conference on Software engineering* (Sorrento, Italy, 1994), 279–287.

[24] Pressman, R. 2009. *Software Engineering: A practitioner's approach*. McGraw-Hill.

[25] Santana, F. et al. 2011. XFlow: An Extensible Tool for Empirical Analysis of Software Systems Evolution. *Proceedings of the VIII Experimental Software Engineering Latin American Workshop* (Rio de Janeiro, Brazil, 2011).

[26] Seaman, C. and Guo, Y. 2011. Chapter 2 - Measuring and Monitoring Technical Debt. M.V. Zelkowitz, ed. Elsevier. 25–46.

[27] Wilkie, F.G. and Kitchenham, B.A. 2000. Coupling measures and change ripples in C++ application software. *J. Syst. Softw.* 52, (Jun. 2000), 157–164.

[28] Yau, S.S. et al. 1978. Ripple effect analysis of software maintenance. *The IEEE Computer Society's Second International Computer Software and Applications Conference* (Nov. 1978), 60–65.