Analysis of Coupling Evolution on Open Source Systems

Bruno L. Sousa
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
bruno.luan.sousa@dcc.ufmg.br

Mariza A. S. Bigonha
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
mariza@dcc.ufmg.br

Kecia A. M. Ferreira
Department of Computing
Federal Center for Technological Education of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
kecia@decom.cefetmg.br

ABSTRACT

Software evolution is an intrinsic process of software life cycle. The comprehension of this process is a central research topic in Software Engineering. It is widely accepted that as a software system evolves, its internal quality declines and its complexity increases. However, there is a gap in the comprehension on how this process occurs in a fine-grained view. In this work, we apply a software metric approach to investigate how the internal quality of object-oriented software systems evolves in the aspect of coupling. More specifically, we analyze (i) how the coupling behavior may be described over the software evolution, (ii) how the coupling behavior affects the reusability and complexity of the systems, and (iii) the percentage of classes from the systems that directly impacts on the coupling evolution. The results and observations of this study are compiled in eight properties of coupling evolution, among which stand out: (i) the coupling behavior is better modeled by a cubic function, (ii) the coupling evolution tends to increase the complexity of the systems, (iii) the systems tend to be designed with a high level of complexity, and (iv) the coupling evolution is affected by a small group of classes.

KEYWORDS

coupling, open source, object-orientation, software evolution, software metrics, software quality

ACM Reference Format:
Bruno L. Sousa, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. 2019. Analysis of Coupling Evolution on Open Source Systems. In *SBCARS '19: XIII Brazilian Symposium on Software Components, Architectures, and Reuse, September 23–27, 2019, Salvador, BA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Software evolution is a process of developing, maintaining, and updating software systems [26]. Such process is essential during the software life cycle because it allows to include or enhance features in the system. In contrast, this process promotes several changes in the software structure requiring that the internal and external quality attributes to be continuously modified.

Lehman [26] analyzed the evolution of large and mature software and proposed seven laws, which are widely known as Lehman's laws. Such study is one of the landmarks on software evolution and it has inspired other works to investigate this topic. For instance, some studies have investigated the applicability of the Lehman's laws in software [17, 21, 24, 29, 35, 45], while others [2, 8, 15, 16, 20, 23, 43] have analyzed the evolution of software growth and characterized the evolution of its internal structure. Although software evolution has been intensely studied, some subjects are still opened [30]. There is a gap in the comprehension on how software evolution occurs in a fine-grained view. For instance, it is widely accepted that as a software system evolves, its internal quality declines and its complexity increases, however it is not known whether there is a pattern to such degradation.

Coupling is the level of dependence among the modules of a software system; it is a central dimension of software modularity and, hence, of software internal quality [34]. In this work, we apply a software metric approach to investigate how the internal quality of Java software systems evolves in the aspect of coupling. So, we carry out an exploratory study on coupling evolution in 10 open source Java systems. The main goal of this study is to investigate how the coupling evolves during the software life cycle. More specifically, we aim to (i) analyze how the coupling behavior may be described over the software evolution, (ii) evaluate how the coupling behavior affects the reusability and the complexity of the software, and (iii) analyze the portion of system classes that directly interfere in the coupling evolution behavior. We have analyzed fan-in and fan-out values because these metrics may efficiently quantify the coupling and they allow to study both the input and output couplings [40]. We propose a method composed of two phases to analyze coupling evolution via software metrics. The first one consists of applying linear regression in the global fan-in and fan-out values from the systems to identify the function that better explains the coupling evolution behavior. In the second phase, we analyze the fan-in and fan-out values of each systems component to identify those that directly interfere in coupling growth or decrease.

The results of this study let to several observations, which we compile in eight properties that describe the coupling behavior. Among these properties, those that stand out are: (i) the coupling growth pattern is better modeled by a cubic function; (ii) the coupling evolution tends to increase the systems complexity; (iii) systems tend to be designed with a high complexity level; and (iv) the coupling evolution behavior is affected by a small group of classes from the systems. The findings being related in this paper contribute to the body of empirical knowledge on open source

SBCARS '19, September 23–27, 2019, Salvador, BA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

software evolution with eight different properties that detail and explain the coupling evolution process in open-source software. Besides, practitioners and researchers may use these properties as a background for understanding the open source software evolution process, and as starting point to propose techniques and methods that improve the internal quality of these systems during their evolution process.

2 RESEARCH METHOD

This section describes the methodology we have applied in this paper.

2.1 Research Questions

We report an empirical study where time series from fan-in/fan-out measures are analyzed to understand and describe the coupling evolution in Java software systems. Fan-in and fan-out are software metrics used to measure coupling in object-oriented software. They indicate the number of references made to a given class by other classes and the number of calls made by a given class to other classes, respectively [24, 40]. There are other metrics that compute the coupling in object-oriented software, such as CBO (coupling between objects) [6], RFC (response for class) [6], COF (coupling factor) [1], MPC (message passing coupling) [27], DAC (data abstraction coupling) [27], and ICP (information-flow-based coupling) [25]. However, COF just computes the global system coupling; RFC, ICP, and MPC are based only on method invocations; DAC are based on only class attributes, and CBO is based on both method invocations and class attributes, but it does not differ the input and output component coupling. Therefore, we decided to use fan-in and fan-out metrics because they consider both method invocations and class attributes as coupling, they allow to measure each internal software component, and they allow to analyze the coupling in both input and output aspects.

The general research question being investigated in this work is:

RQ1. How does coupling evolve in open source Java software systems?

This research question aims to study the coupling evolution in open source Java systems and extract properties that describe its behavior over the software life cycle. To detail RQ1, we subdivide it into three other specific research questions, as follows.

RQ1.1. What kind of model better represents the coupling growth pattern in open source Java software systems?

RQ1.2. How does the relation between fan-in and fan-out behave throughout the evolution in open source Java software systems?

RQ1.3. What is the percentage of classes contained in the software that direct interfere in the coupling growth/decrease?

2.2 Dataset

We have used a public dataset, `COMETS (Code Metrics Time Series)`, which is composed of time series from 17 well-known software metrics regarding 10 open source Java systems [7]. These time series were collected in intervals of bi-weeks, i.e., 14 days [7]. Table 1 summarizes the `COMETS` dataset, presenting its main features.

We identified other three dataset in the literature: `D'Ambros dataset` [11], `Helix` [39], and `Qualitas Corpus` [42]. However, `Qualitas Corpus` does not provide time series from object-oriented metrics, `Helix` does not include time series on coupling metrics, and `D'Ambros dataset` even provides time series on coupling metrics, but `COMETS` has a greater number of systems and versions than `D'Ambros dataset`. So, we include `COMETS` because it is the largest dataset with time series regarding coupling metrics that we have identified in the literature.

2.3 Behavior Analysis

This section describes the first phase of our methodology, where we investigate what type of model better explains the coupling growth pattern over its evolution. This phase consists of five steps.

Step 1. Normalizing the fan-in and fan-out time series to obtain a global measure for each one of these two metrics, and then, model and evaluate the systems global coupling. As the systems has a set of time series, we use the sum of fan-in/fan-out values to represent their global measure. So, for each system we separately sum the fan-in/fan-out values of all classes in each version, and obtain the global time series for each metric

Step 2. Applying linear regression methods in the global time series of fan-in/fan-out to define the analyzed models. Other authors have used ARIMA to model software metrics evolution [38]. But to use this approach it is necessary to collect the time series over a well-defined time scale, such as days, months, or years. Initially, we tried to apply ARIMA to model the coupling time series. However, after applying it, we have identified that it was not applicable to our data since it presents a random walk, i.e., a type of model that describes a random behavior of the data. Then, we decide to use linear regression and model the time series in terms of versions, where each version corresponds to a two-week interval. We model both fan-in and fan-out using the following models: (i) linear, (ii) quadratic (polynomial at degree 2), (iii) cubic (polynomial at degree 3), and (iv) logarithmic. We have used all these models to evaluate which of them better explains the fan-in/fan-out evolution. It is important to highlight that although we use models to characterize the coupling growth pattern, our purpose is not to provide a prediction model.

Step 3. Regression methods require that the assumption of independence be satisfied to ensure the validity of the models [5]. However, it is common the emergence of error terms autocorrelated when we use time series. If the autocorrelation is not treated, the estimates of coefficients and their standard errors will be wrong and the model will not correctly describe the time series behavior [5]. So, in this step we evaluate the errors of each model and remove the occurrences of autocorrelation in their error terms. To identify the existence of autocorrelation, we apply a statistic test named Durbin-Watson test [13], which detects the presence of autocorrelation at lag 1 in the residuals from a regression analysis. Autocorrelation at lag 1 in time series means that an observation x_i is dependent on the observation x_{i-1}. When identified, we have to remove it from the model residuals via autoregression [9] and include the autoregressive error coefficient in the generated models.

Step 4. To evaluate the models, we compute their adjusted determination coefficient ($\overline{R}^2$). Both determination coefficient (R^2) and adjusted determination coefficient ($\overline{R}^2$) are metrics derived from the analysis based on the general linear model, which measure

Table 1: Systems of the COMETS dataset.

#	System Name	Description	Time Frame	# Versions
1	Eclipse JDT Core	Compiler and other tools for Java	07/01/2001 - 06/14/2008	183
2	Eclipse PDE UI	Set of tools to create, develop, test, debug and deploy Eclipse plug-ins, fragments, features, update sites and RCP products	06/01/2001 - 09/06/2008	191
3	Equinox Framework	OSGi application implementor	01/01/2005 - 06/14/2008	91
4	Hibernate Core	Database persistence framework	06/13/2007 - 03/02/2011	98
5	JabRef	Bibliography reference manager	10/14/2003 - 11/11/2011	212
6	Lucene	Search software and document indexing API	01/01/2005 - 10/04/2008	99
7	Pentaho Console	Software for business intelligence	04/01/2008 - 12/07/2010	72
8	PMD	Source code analyzer	06/22/2002 - 12/11/2011	248
9	Spring Framework	Java application development framework	12/17/2003 - 11/25/2009	156
10	TV-Browser	Electronic TV guide	04/23/2003 - 08/27/2011	221

the degree of adjust of the model to the data in order to understand to what extent the model explains the analyzed data [31]. We chose the ($\overline{R}^2$) instead of R^2 because it takes into account the number of parameters in the model and penalizes the inclusion of less explanatory parameters.

Step 5. Testing the statistic significance of the differences between the $\overline{R}^2$ values of the models. We use the Wilcoxon signed-rank test to evaluate this significance. Wilcoxon signed-rank test is a non-parametric statistical hypothesis test, whose purpose is to compare two related or matched samples or repeated measurements on a single sample to evaluate if their populations mean ranks differ [10]. We use it because the difference values between the $\overline{R}^2$ scores are not normally distributed. We consider the following hypothesis during the application of the test:

- **H_0:** the difference between the pairs follows a symmetric distribution around zero
- **H_1:** the difference between the pairs does not follow a symmetric distribution around zero

To reject the null hypothesis (H_0), we consider a p-value less than 0.01. After applying it, we analyze the $\overline{R}^2$ scores of the models to identify the one that better represents the coupling growth pattern.

2.4 Trend Analysis

This section describes the second phase of our methodology, where we analyze the percentage of classes that directly interfere in both coupling growth and decrease. This phase consists of seven steps.

Step 1. Couto et al. [7] have included -1 value in the cases where classes are not present in a given version. As these values may bias our results, in this step we remove them, and reorganize the other values in the time series.

Step 2. Software systems undergo several modifications over their life cycle, and consequently, a class may appear or disappear any time. After analyzing the time series of the systems, we have identified that some classes appear and disappear more than once during their life cycle. When this situation occurs, the time series is broken in several small sub-series where some observations do not have value. We have classified classes with this behavior as "ghost" classes. Figure 1 shows and example of fan-in time series of a ghost class extracted from `Eclipse JDT Core`. This class is removed from the system, and after some versions, it is inserted into the system again. This situation may bias the results. So, as the ghost classes make up a small part of the systems, no more than 2% of all classes, we decided not analyzing them in this part of the study.

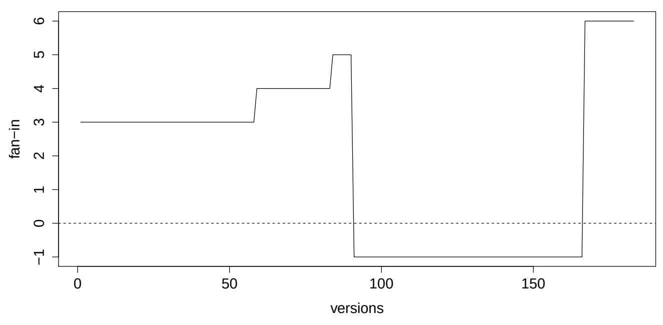

Figure 1: Time series of a ghost class.

Step 3. Applying trend tests in the time series to analyze and identify the percentage of classes that affect the coupling growth and decrease in the systems. We have used three different tests: (i) Mann-Kendall [22], (ii) Cox-Stuart [32], and (iii) Wald-Wolfowitz [32]. We use them because they are pointed out as useful for trend identification in time series [32]. As they are based on hypothesis analysis to check the presence of trend in the time series, we consider the following hypothesis:

- **H_0:** there is no trend in the time series
- **H_1:** there is a trend in the time series

Step 4. Before applying Mann-Kendall test, we have analyzed the time series to check if the observations were independent or they were autocorrelated. To simplify our analysis, we apply trend tests without checking autocorrelation in the time series. The Mann-Kendall test is run in its original version. To define the existence of trend in time series, we define the following criteria: *"time series has a growth or decrease trend if, and only if, the null hypothesis is rejected at least in two of the three tests"*. However, as we did not check autocorrelation at first, the result of the Mann-Kendall test may bias the results. To overcome this problem, we have analyzed the trend results by looking for "doubtful" cases, which appear when:

Case 1 : there are two trend tests where the null hypothesis is rejected, and one of them is the Mann-Kendall test

Case 2 : there is just one trend test where the null hypothesis is rejected, and necessarily it is not the Mann-Kendall test

After identifying the "doubtful" cases, we plot and analyze their autocorrelation and partial autocorrelation charts. In the cases where we identify presence of autocorrelation, we run the Mann-Kendall test again with the Block Bootstrap approach in Step 5.

Step 5. Running the Mann-Kendall test with the Block Bootstrap approach for the "doubtful" cases found in Step 4. Block Bootstrap method is a robust and flexible resampling approach used to estimate the significance of a statistical test [12, 14, 33, 47]. This method is used in conjunction with other statistical tests for trend removal and autocorrelation in the data. It shuffles the elements of the data a lot of times by using replacements and obtains many other samples with the same number of elements as the original data [36]. After obtaining the samples, the statistical test is calculated for each sample and the distribution probability is estimated. At the end, the method computes the statistical analysis for the original data and compares its results with the distribution generated for the samples to estimate the level of significance of the test. As the data used by this method are serially dependent, it runs the whole process in blocks so that the autocorrelation may be replicated. During the implementation of this approach, we have to inform some parameters such as number of samples to be generated by the Block Bootstrap in each case (R) and the size of the blocks (L). In general, set up these parameters is one of the main problems of this approach because they depend on the number of observations existing in the time series. However, some studies have analyzed these parameters to establish values that better fit to them. For instance, Svensson et al. [41] find that good stability in significance level estimates may be obtained with 2,000 samples. Moreover, Önöz and Bayazit [36] analyze values of the block size and show that five is the size that better fits as a general value, and minimizes the probability of errors occurrence. So, based on these studies, we implement the Block Bootstrap approach with R and L parameters being 2,000 and 5, respectively.

After implementing this approach, we generate a p-value as the final value of this approach for each time series and we analyze them considering the hypothesis of the original Mann-Kendall test. To reject the null hypothesis, we consider a p-value less than 0.05.

Step 6. Analyzing the p-value of the three trend tests and apply the criteria defined in Step 4. For all time series we identify the presence of autocorrelation, we analyze the p-values resulting from the Mann-Kendall test with the Block Bootstrap approach, Cox-Stuart, and Wald-Wolfowitz. For the time series we do not identify presence of autocorrelation, we analyze the p-values resulting from the Mann-Kendall test without the Block Bootstrap approach, Cox-Stuart, and Wald-Wolfowitz.

Step 7. Evaluating the type of trend in the systems classes. So, we plot the original time series chart with a trend line to visually characterize the type of trend. After that, we manually classify the trends by considering the criteria as follows.

- **Upward trend:** it is a pattern whose distance between the trend line and x-axis increases over the x-axis
- **Downward trend:** it is a pattern whose distance between the trend line and x-axis decreases over the x-axis
- **Undefined trend:** it is a pattern whose distance between the trend line and x-axis remains the same over the x-axis

3 OBSERVATIONS ON THE COUPLING EVOLUTION

This section presents some observation regarding the coupling evolution by answering the specific research questions.

3.1 Coupling Evolution in the System Level

This section answers RQ1.1.

RQ1.1. What kind of model better represents the coupling growth pattern in open source Java software systems?

This research question aims to identify how the coupling growth pattern may be described in terms of fan-in and fan-out during the Java systems evolution. To answer RQ1.1, we apply regressions techniques on the global time series regarding the 10 systems from COMETS, and compute the adjusted determination coefficient ($\overline{R}^2$) to evaluate and compare the resulting adjust of these models and identify the one that better describes the coupling growth pattern.

Table 2 summarizes the $\overline{R}^2$ scores computed for the models extracted to both fan-in and fan-out metrics in each system. The "lin.", "quad.", "cub.", and "log." columns indicate the $\overline{R}^2$ scores from the linear, quadratic, cubic, and logarithmic models, respectively.

After computing the $\overline{R}^2$ scores of the models, we apply the Wilcoxon signed-rank test to check if the difference between them has statistic significance. To test the significance, we combine the models in pairs, and consider the hypothesis indicated in Step 5 in Section 2.3. To reject the null hypothesis, we consider a confidence of 99%, i.e., a p-value less than 0.01. Table 3 summarizes the p-values obtained after applying Wilcoxon signed-rank test. In this table, the models linear, quadratic, cubic, and logarithmic are named as "lin.", "quad.", "cub.", and "log.", respectively.

The returned p-values by the test show the $\overline{R}^2$ scores of the models presenting significant differences, except for one combination: linear and logarithmic models (case 3 in Table 3). In this case, the null hypothesis may not be rejected at 99% confidence, and therefore, we may not say anything about the difference of $\overline{R}^2$ between these two models.

After comparing the results of the Wilcoxon signed-rank test, we analyze the $\overline{R}^2$ scores of the models to identify which model better describes the coupling growth pattern. We conclude that both fan-in and fan-out have a growth pattern that is better explained by a cubic-order model. However, although fan-in and fan-out growth pattern may be represented by the same model, when we compare the global time series, we noticed that fan-out is extremely greater than fan-in. Figure 2 shows this situation via a line chart from the global fan-in and fan-out time series.

Summary of RQ1.1. In response to RQ1.1, we identify that both fan-in and fan-out have a growth pattern that may be better explained by a cubic-order model. However, although the same model better explains both fan-in and fan-out growth pattern, we

Table 2: $\overline{R}^2$ values computed from the fan-in and fan-out models.

System	Fan-in				Fan-out			
	lin.	quad.	cub.	log.	lin.	quad.	cub.	log.
Eclipse JDT Core	0.973	0.989	0.992	0.908	0.978	0.993	0.994	0.907
Eclipse PDE UI	0.989	0.994	0.996	0.803	0.990	0.995	0.998	0.809
Equinox Framework	0.971	0.975	0.979	0.901	0.978	0.988	0.988	0.919
Hibernate Core	0.912	0.968	0.971	0.677	0.902	0.969	0.973	0.645
JabRef	0.932	0.988	0.995	0.971	0.921	0.988	0.996	0.971
Lucene	0.933	0.939	0.951	0.766	0.930	0.931	0.957	0.804
Pentaho Console	0.406	0.685	0.919	0.506	0.695	0.901	0.975	0.939
PMD	0.976	0.977	0.986	0.837	0.976	0.976	0.986	0.831
Spring Framework	0.842	0.995	0.996	0.928	0.770	0.997	0.997	0.938
TV-Browser	0.921	0.997	0.997	0.937	0.871	0.992	0.994	0.948

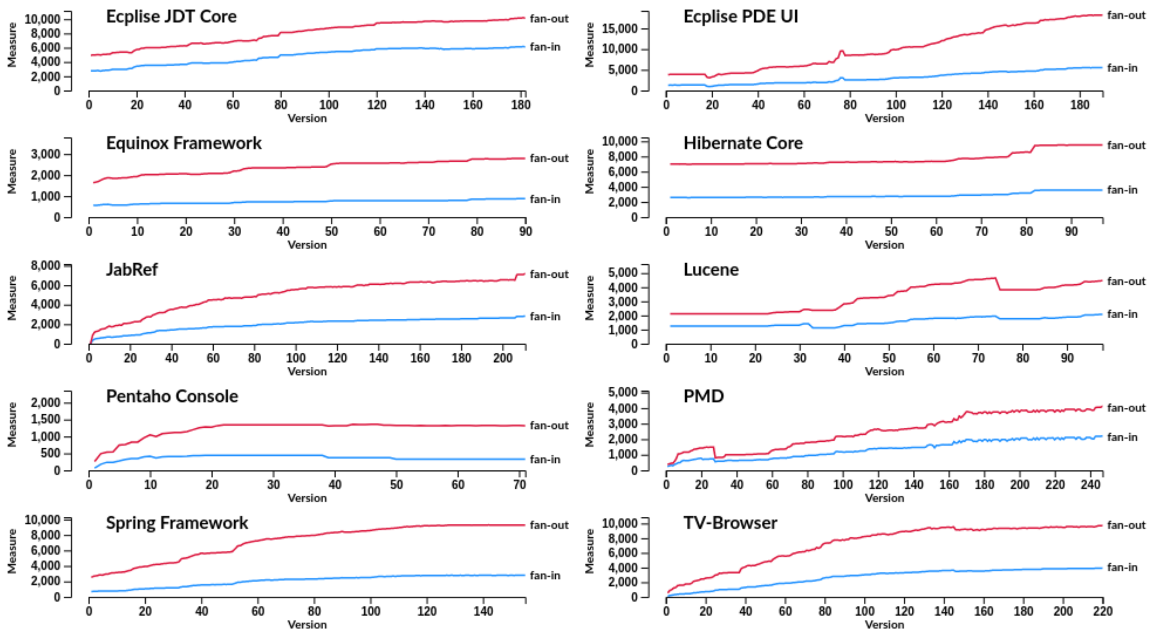

Figure 2: Global fan-in/fan-out time series of the analyzed systems.

Table 3: Wilcoxon signed-rank test p-values

#	Combinations	Fan-in p-values	Fan-out p-values
1	lin. *versus* quad.	0.00195	0.00195
2	lin. *versus* cub.	0.00195	0.00195
3	lin. *versus* log.	0.19336	0.55664
4	quad. *versus* cub.	0.00195	0.00977
5	quad. *versus* log.	0.00195	0.00586
6	cub. *versus* log.	0.00195	0.00195

identify an extreme difference between these two metrics since the fan-out values are much greater than fan-in values.

3.2 Evolution of Fan-in/Fan-out Relation

This section answers RQ1.2.

RQ1.2. How does the relation between fan-in and fan-out behave throughout the evolution in open source Java software systems?

This research question aims to analyze how the relation between fan-in and fan-out impacts on the coupling evolution in open source Java software systems. In a previous study, Berard [3] categorizes the occurrence of the coupling in software as two types: necessary and unnecessary. Basically, necessary coupling consists of high fan-in and low fan-out and unnecessary coupling consists of high fan-out and low fan-in [3]. Moreover, according to Lee et al. [24], high fan-in, i.e. necessary coupling, usually represents a good object design and high level of reuse, since classes at the same package are reused together. In contrast, high fan-out, i.e. unnecessary coupling, is not desirable in software because it is an indication of complexity, and low reusability [3, 4, 19, 28]. In this way, we use the definition of necessary and unnecessary coupling, given by Martin [28], to answer this research question.

After identifying that fan-in and fan-out follow the same growth pattern, we analyze which type of coupling stands out during the systems evolution. So, we compute the necessary and unnecessary coupling rates for each systems version. The necessary coupling rate consists of dividing fan-in by fan-out, and the unnecessary coupling rate consists of dividing fan-out by fan-in. Figure 3 summarizes the behavior of these two types of coupling.

Analyzing Figure 3, we observe that for most of systems, the unnecessary coupling rate increases over their evolution, whereas the necessary coupling rate grows minimally remaining almost stable. This scenario is in line with Lehman's 2^{nd} and 6^{th}, which indicate increasing complexity and declining quality in the systems during their evolution. Moreover, we observe that for almost all systems, except to JabRef, the unnecessary coupling rate is much greater than the necessary coupling rate in the first systems version. This finding shows that most open source systems tend to be developed for the first time already with a high degree of complexity.

Summary of RQ1.2. The unnecessary coupling rate tends to increase over the systems evolution. Besides, the systems have a high unnecessary coupling rate since their first version. The necessary coupling has a slight growth over time. This behavior shows that, in general, systems are created with high complexity and low quality and the complexity tends to be even higher over the time, whereas the quality tends to be even low.

3.3 Coupling Growth/Decrease Analysis

This section answers RQ1.3.

RQ1.3. What is the percentage of classes contained in the software that direct interfere in the coupling growth/decrease?

To answer RQ1.3, we identify the classes responsible for increasing and decreasing the coupling. Then, we carry out a trend analysis in the time series of the systems classes and computed the percentage of classes whose fan-in and fan-out have increased/decreased over time. Figure 4 presents the distribution of percentage of classes with fan-in growth, fan-in decrease, fan-out growth, and fan-out decrease, and Table 4 provides a descriptive analysis in terms of percentiles of the boxplot in Figure 4. We separate the discussion about the results regarding coupling growth and decrease.

Table 4: Descriptive Analysis of the Percentage Distribution of Classes within the Systems that Impact on Coupling Growth/Decrease.

Event	0%	25%	50%	75%	100%
Fan-In Growth	3.00	8.00	8.50	12.75	23.00
Fan-Out Growth	5.00	6.50	11.50	17.50	27.00
Fan-In Decrease	1.00	2.25	3.00	3.75	4.00
Fan-Out Decrease	3.00	6.25	7.50	9.00	10.00

3.3.1 Classes Responsible for Coupling Growth. Figure 4 and Table 4 show that the median of percentage of classes in the systems responsible for "fan-in growth" and "fan-out growth" are 8.50% and 12.75%, respectively. The maximum percentages we have found were 23% and 27% for fan-in and fan-out, respectively. Nevertheless, it is important to highlight that 23% is as an outlier in Figure 4 because only one of the system in analyze presents this value.

The results for fan-in/fan-out coupling show that although they have a growth pattern that is better described by a cubic-order model, their growth are directly influenced by a small group of classes that represents no more than 30% of the systems. We also analyze how these classes are distribute over the systems versions and we identify that, in 50% of the systems, more than 25% out of the classes that contribute to both fan-in and fan-out growth are classes introduced in the first version of the system and not removed during their evolution.

3.3.2 Classes Responsible for Coupling Decrease. Observing the results shown in Figure 4 and Table 4, we see that the median of percentage of classes responsible for "fan-in decrease" and "fan-out decrease" are 3% and 7.5%, respectively. The maximum percentages are 4% and 10% for fan-in and fan-out, respectively. Therefore, this result shows that there are a low percentage of classes, no more than 10%, that impact on the coupling decrease. Besides, there is a significant discrepancy between the results obtained for growth and decrease coupling. We also analyze the distribution of the classes that impact on the coupling decrease. Just as in coupling growth analysis, we identify that in 50% of the systems, legacy classes represent more than 25% out of the total of these classes.

3.3.3 Growth versus Decrease. Here, we perform an intersection of the trend results for fan-in and fan-out to identify the percentage of classes that have the following behaviors: (i) both fan-in and fan-out growth, (ii) both fan-in and fan-out decrease, (iii) fan-in growth and fan-out decrease, and (iv) fan-in decrease and fan-out growth. Table 5, presents the percentages obtained for these cases.

Table 5: Intersection of the trend results for fan-in and fan-out.

System	i	ii	iii	iv
Eclipse JDT Core	15%	1%	2%	1%
Eclipse PDE UI	4%	1%	1%	0%
Equinox Framework	3%	0%	1%	1%
Hibernate Core	1%	0%	0%	0%
JabRef	5%	0%	1%	1%
Lucene	2%	0%	1%	1%
Pentaho Console	1%	0%	1%	0%
PMD	3%	1%	0%	0%
Spring Framework	5%	1%	1%	1%
TV-Browser	10%	1%	1%	1%

The behavior of Case i have the highest chance to occur since it has the maximum percentages, 15% and 10% respectively, and present a greater percentage than the other cases in all systems. Even though, when we analyze the distribution of the percentages in the Case i , we notice that both 15% and 10% are outliers, and the median and maximum values are 3.5% and 5%, respectively, by disregarding these two outliers. So, in general, we observe a low percentage of classes that follow the patterns being analyzed in all cases. This result shows that most of the classes that directly

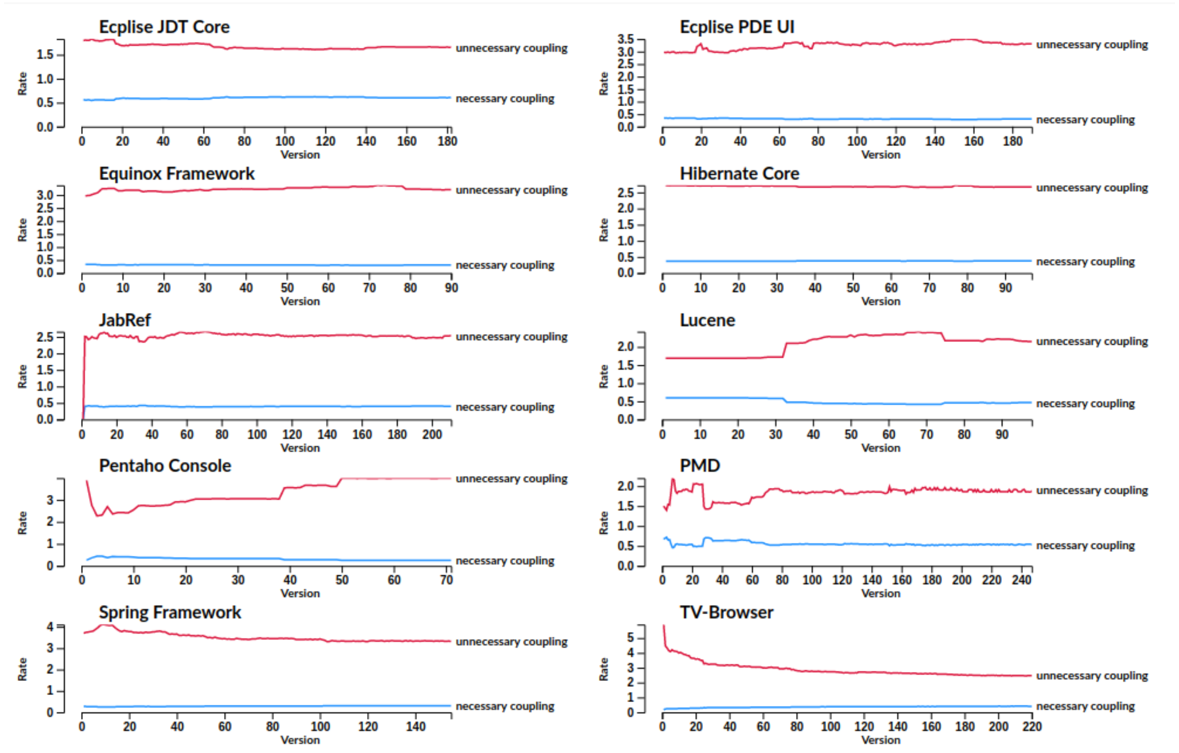

Figure 3: Evolution of unnecessary and necessary coupling.

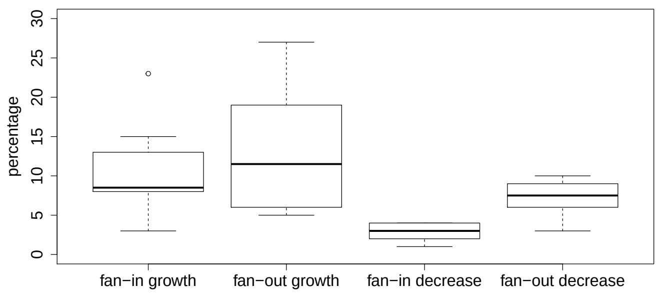

Figure 4: Percentage distribution of classes within the systems that impact on coupling growth/decrease.

impact the coupling evolution do not tend to follow a combined pattern in terms of growth and decrease of fan-in and fan-out.

Summary of RQ1.3. Coupling evolution is affected by a small and specific group of classes in the systems. When we analyze the distribution of these classes over the systems versions, we identify a strong influence of legacy classes on the coupling growth and decreased in 50% of the analyzed systems. We consider a legacy class in this analysis as being the one that is introduced in the first version of a system and it is not removed during its evolution. Moreover, the growth/decrease of fan-in, and fan-out of the classes that directly impact the coupling evolution are not associated.

4 COUPLING EVOLUTION PROPERTIES

This section compiles the observations of coupling evolution, and synthesizes our results as properties of coupling evolution to answer and detail the general research question (RQ1). In each property we present a brief description about it. The eight coupling evolution properties identified in this work are summarized as follows.

Growth pattern modeled by a cubic function. The growth pattern of coupling evolution is better modeled by a cubic function since it has a flexible curve that may identify the small variations in the global time series of the systems.

Lehman's 2^{nd} and 6^{th} laws are applied. The Lehman's 2^{nd} and 6^{th} laws indicate that the complexity tends to increase and the quality tends to decline during the software evolution. When we perform a quality analysis of the coupling evolution, these two behaviors described by these laws are detected.

Unnecessary coupling is greater than necessary coupling. Necessary coupling consists of high fan-in and low fan-out, whereas unnecessary coupling consists of low fan-in and high fan-out. Comparing the rate of these two types of coupling over the evolution, we notice that there is a big difference between them, with the unnecessary coupling always greater than necessary coupling.

Fast unnecessary coupling growth and slow necessary coupling growth. Comparing the unnecessary and necessary coupling evolution, in the most of the systems the unnecessary coupling presents a rate that tends to grow up fast and the necessary coupling shows a rate that tends to grow up slowly almost remaining stable over the evolution of the systems.

Complexity is introduced into the first software version. Coupling is divided as unnecessary and necessary, and while the necessary coupling favors the reusability, the unnecessary coupling favors the complexity [3, 4, 19, 24, 28]. We notice that in the first systems version the unnecessary coupling rate is extremely greater than the necessary coupling rate. Based on this analysis and the quality indication pointed by the literature, we conclude that complexity is introduced in systems since their initial version.

A small group of classes influences the coupling. There is a small group of classes in a system that directly influences the coupling growth and decrease, equivalent to no more than 30% and 10% of the total classes, respectively.

Legacy classes mainly contribute to the coupling evolution. Legacy classes are classes introduced in the first system version that are not removed during its evolution. Analyzing the distribution of the classes that influence the coupling growth/decrease over the systems versions, we identify a strong presence of legacy classes on 50% of classes that make up the systems. Therefore, we have extracted this property since classes with this role may have a high probability of directly affecting the coupling growth or decrease.

The evolution of fan-in and fan-out are not associated. We do not identify pattern in terms of growth and decrease of fan-in and fan-out for the classes that impact on the coupling evolution. This means that the growth/decrease of fan-in is not related to the growth/decrease of fan-out, i.e., when a class becomes more/less dependent of other classes does not necessarily imply that the other classes will demand more/less services from it.

5 THREATS TO VALIDITY

This section presents the main threats to validity according to the guidelines proposed by Wohlin et al. [44].

Internal Validity. This kind of validity is concerned with the risk of some factor to affect the investigation of a causal relation between two variable of the experiment [44]. In our trend analysis, we have used statistical tests for identifying trend in time series, and finding the classes that directly affect the coupling growth/decrease in the systems. The choice of the tests may be considered a threat to validity since they are not able to ensure their results are error free. To mitigate this threat, we evaluate each time series with three different tests and determined the presence of trend if and only if it is confirmed at least two out of three trend tests. Besides, the tests used to check the existence of trends in time series are point out as useful and reliable.

External Validity. This kind of validity is concerned with to what extent it is possible to generalize the findings, and to what extent they are of interest to other people outside the investigated case [44]. We have presented an exploratory study regarding the coupling evolution in open source Java systems. We have used a dataset composed of time series from 10 relevant systems. Although the obtained results provide relevant findings regarding the coupling evolution, such observations and properties may not be generalized to other system domains, such as proprietary software and systems written in any language other than Java.

Construct Validity. This kind of validity is concerned with to what extent the experiment setting reflects the theory that the researcher has in mind [44]. To study the coupling evolution, we chose some metrics that measure and quantify this aspect. However, the choice of these metrics may be considered a threat to validity since it may result in metrics that do not provide a good representation of the coupling. To mitigate it, we chose two well-known and very used metrics, fan-in and fan-out, to measure the coupling. Besides, fan-in and fan-out allow evaluating the coupling in two viewpoints: input and output. Such a distinction of coupling is a factor that differs them from the other coupling metrics that usually measure the coupling considering these characteristics as only one.

During the trend analysis, we carry out a step where we removed the "ghost" classes from our analysis because they produce broken time series, which does not present values for some observations. This step may be considered a threat to validity, since these classes may indicate relevant information about the coupling growth/decrease trend. To mitigate it, we analyze them separately to check if they have significant information. We identify that the removal of the "ghost" classes does not bias our results since they make up a small portion of the systems, no more than 2% of all classes, and they have a random and inconclusive trend pattern.

Conclusion Validity. This kind of validity is concerned with to what extent the data and analysis are dependent on the specific research. When analyzing the coupling growth pattern, we have used linear regression techniques to create models that describe this pattern. Although linear regression is a method usually used to define models, it may bias the results when the data are serially dependent. Therefore, to mitigate this threat, after applying the linear regression methods, we analyze the models residuals, and for the cases that presented autocorrelation, we use an auto-regression technique in their residuals to model their errors and remove the presence of this autocorrelation.

To evaluate the quality of the models and identify the one that better describes the behavior of the coupling growth pattern, we use a statistical test to check if the differences between the determination coefficients of the models are significant. However, the choice of the test may be considered a threat to validity since tests with low statistical power may be chosen. Therefore, to mitigate this threat, we use the Wilcoxon signed-rank test since it is well known and usually used for other studies in the literature [23].

6 RELATED WORK

This section provides an overview of some related work and discusses the main differences between them and our study.

Software evolution has been extensively studied in the last years. One of the great landmarks in this topic was the creation of software evolution laws in the '70s by Lehman and contributors [26]. Since then, several studies have studied the application of these laws to open source systems. For instance, Godfrey and Tu [17] study the Linux Kernel and some other systems and found that the Linux has a continuous growth in a high rate and many open source systems do not follow some Lehman's laws. Lee et al. [24] analyze the JFreeChart system, and identify that it follows the 1^{st}, 2^{nd} and

6^{th} laws, but the 7^{th} law is not applicable in it because its quality tends to better over time. Mens et al. [29] study the Lehman's laws in Eclipse evolution with support of number of errors, number of removed/ added files, and number of changed files metrics. The results indicate evidence of increasing complexity and continuous growth, which complies with the 2^{nd} and 6^{th} laws, respectively. Besides, they also identify a high occurrence of changes, which shows that Eclipse also follows the 1^{st} software evolution law.

Xie et al. [45] analyze the evolution of open source systems to check if they obey the Lehman's laws. They find that open source systems follow some rules, such as: continuing change (1^{st}), increasing complexity (2^{nd}), self-regulation (3^{rd}), and continuing growth (6^{th}). They also realize that most of the modifications occur in a small portion of source code, and changes usually occur in the initial phases of the software evolution. Israeli and Feitelson [21] evaluate the Linux kernel to check if its evolution reflects the Lehman's laws. They conclude that most laws occur, but the complexity decreases over time when several small functions are added in the software. Oliveira et al. [35] study a software product line (SPL) to understand its evolution. They find that the 4^{th} Lehman's law is completely supported and the 5^{th} and 6^{th} Lehman's laws are partially supported by SPL.

Besides the Lehman's laws, software evolution has been studied under the growth aspect. Herraiz et al. [20] characterize the growth evolution of open source systems by comparing the evolution of the following metrics: source lines of code and number of modules/files. They find that these two metrics follow the same behavior over time, and the patterns that are not conforming to Lehman's law are indeed apparent. Koch [23] analyze the evolution of many open source systems to understand how they evolve and describe their evolutionary behavior, and find that the growth rate of open source systems is linear or decrease over time, but there is a significant percentage of projects that have a super-linear growth. He also identifies that systems with super linear growth are impacted by the number of participants and inequality in the distribution of work within the development team. Adnan [2] studies an open source Java system to characterize it in terms of size of methods and size of classes. He notes that the system has a sub-linear growth pattern, and its evolution is explained by some of Lehman's laws, especially the 1^{st} law.

Some studies in the literature have characterized the software evolution by analyzing the internal structure of the systems. Ferreira et al. [15] analyze the software evolution by applying complex networks concepts and find that the connectivity decreases as the software grow, and classes that provide services for many other classes are unstable and their cohesion degrade over time. Besides, they provide a macroscopic model of the software structure that they named as Little House. In a latter study, Ferreira et al. [16] model and characterize the internal structure of 13 object-oriented programs using the Little House model. They find that classes that fit in the LSCC and Out components of this model are more critical, and they tend to suffer substantial degradation during their evolution. Trindade et al. [43] investigate the evolution and growth of the internal structure from object-oriented systems, and find that their architecture follows an evolution pattern. They analyze this pattern and define a stochastic model that explains this evolution pattern. Ostrand et al. [37] develop a statistical prediction model based on history information from a large industrial inventory control system to predict faults occurrence in software. They evaluate the accuracy of this model for two different systems and identify that it correctly predicts 73% and 74% of the faults existing in these two systems. Hamill and Goseva-Popstojanova [18] study faults and failures data to identify where the faults that lead to software failures are localized into the software, and how the type of software failures are distributed in the software. They consider failures as being cases where the software does not behave like it is required, and faults as being accidental conditions in the software operation that may the software to fail. They find that failures are caused by multiple faults spread throughout the system. Besides, requirements faults, coding faults, and data problems are the most common types of faults that occur in software, and the majority them occur in a small portion of the system. Couto et al. [8] analyze the evolution of defects and software metrics in open source Java systems, and propose a defect prediction approach to warn programmers about the probability of defects occurrence. Yang et al. [46] investigate the predictive power of unsupervised models in just-in-time defect prediction, and compare their power with supervised models. They identify that unsupervised models have a good predictive effectiveness and many simple unsupervised models perform better than supervised defect prediction models existing in the literature.

Our work presents an exploratory study on coupling evolution in Java systems. We have analyzed how the coupling behaves during its evolution, how this behavior affects the systems quality, and what is the percentage of the system that directly impacts on the coupling evolution behavior. Most of previous studies have investigated if the Lehman's laws apply to evolution certain software [17, 21, 24, 29, 35, 45]. The present study differs from them because we do not aim to analyze the applicability of Lehman's laws, but how the coupling evolves. Besides, other studies have examined the software growth evolution, characterized the evolution of the internal structure of systems, and propose prediction defect models [2, 8, 15, 16, 18, 20, 23, 37, 43, 46]. Our study differs from them because we analyze the evolution of the complexity instead of growth, we use different metrics, and we evaluate how the coupling evolution impacts the quality instead of internal structure.

7 CONCLUSION

This paper presents an exploratory study on coupling evolution in Java systems. Our main goals are: (i) analyze how the coupling behaves during the systems evolution and how they may be explained, (ii) evaluate how the evolution of coupling behavior affects the systems reusability and complexity, and (iii) identify what is the part of the systems that directly impacts on the coupling behavior.

In this study, we use a dataset composed of time series from software metrics regarding 10 open source Java systems. To quantify the coupling, we analyze time series from fan-in and fan-out metrics because they characterize the coupling in two aspects, input and output of a module, respectively. To achieve our goal, we propose a method composed of two phases. At first, we normalize the data to create time series that represents the global coupling of the systems, and apply linear regression methods to identify which model better explains the coupling evolution behavior in open source systems.

After that, we analyze the times series from the systems classes to identify the ones that directly impact on coupling growth or decrease in the systems.

The results and analysis we perform in this study are compiled in eight properties that describe and summarize the coupling evolution in Java system. Among these properties, those that stand out are: (i) the coupling growth pattern is better modeled by a cubic function, (ii) the coupling evolution tends to increase the systems complexity, respecting the 2^{nd} and 6^{th} Lehman's laws, (iii) systems tend to be designed with a high level of complexity, and (iv) the coupling behavior is affected by a small group of classes, which in general represent no more than 30% of the system classes that influence the coupling growth and no more than 10% of the system classes that influence the coupling decrease. The findings of this work are valuable to the literature on software evolution because they report and detail eight different properties that occurs during the coupling evolution process in open source software. Besides, these properties may serve as a background and object of study to practitioners and researchers in the proposal of techniques and methods that improve the interval quality of the systems during their evolution.

As future work we intend to (i) define a global model to predict how the coupling evolves, (ii) replicate this study for other software aspects and metrics, and (iii) investigate how the evolution of the metrics occurs considering the system from different domains.

8 ACKNOWLEDGE

This work was sponsored by CAPES.

REFERENCES

[1] F. B. Abreu and R. Carapuça. 1994. Object-oriented software engineering: Measuring and controlling the development process. In *ICSQ*, Vol. 186. 1–8.

[2] Sinan Diwan Adnan. 2019. Software Evolution on Azureus Bit Torrent Software: A Study on Growth and Change Analysis. *Journal of Engineering Science and Technology* 14, 1 (2019), 430–447.

[3] Edward V. Berard. 1993. *Essays on Object-oriented Software Engineering (Vol. 1).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[4] Grady Booch. 1991. *Object Oriented Design with Applications.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

[5] B.L. Bowerman and R.T. O'Connell. 1993. *Forecasting and Time Series: An Applied Approach.* Duxbury Press.

[6] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.

[7] Cesar Couto, Cristiano Amaral Maffort, Rogel Garcia, and Marco Tulio Valente. 2013. COMETS: a dataset for empirical research on software evolution using source code metrics and time series analysis. *ACM SIGSOFT Software Engineering Notes* 38, 1 (2013), 1–3.

[8] Cesar Couto, Pedro Pires, Marco Tulio Valente, Roberto S Bigonha, and Nicolas Anquetil. 2014. Predicting software defects with causality tests. *Journal of Systems and Software* 93 (2014), 24–41.

[9] Paul S. P. Cowpertwait and Andrew V. Metcalfe. 2009. *Introductory Time Series with R* (1st ed.). Springer Publishing Company, Incorporated.

[10] Peter Dalgaard. 2008. *Introductory Statistics with R* (second ed.). Springer, New York.

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *MSR 2010.* IEEE, 31–41.

[12] A. C. Davison and D. V. Hinkley. 1997. *Bootstrap methods and their application.* Cambridge University Press, Cambridge, UK.

[13] James Durbin and Geoffrey S Watson. 1951. Testing for serial correlation in least squares regression. II. *Biometrika* 38, 1/2 (1951), 159–177.

[14] Bradley Efron and Robert J. Tibshirani. 1993. *An Introduction to the Bootstrap.* Number 57 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, Boca Raton, Florida, USA.

[15] K. A. M. Ferreira, M. A. Bigonha, R. S. Bigonha, and B. M. Gomes. 2011. Software evolution characterization-a complex network approach. *SBQS* (2011), 41–55.

[16] K. A. M. Ferreira, R. C. N. Moreira, M. A. S. Bigonha, and R. S. Bigonha. 2012. The evolving structures of software systems. In *WETSoM.* 28–34.

[17] Michael Godfrey and Qiang Tu. 2001. Growth, Evolution, and Structural Change in Open Source Software. In *IWPSE.* ACM, New York, NY, USA, 103–106.

[18] Maggie Hamill and Katerina Goseva-Popstojanova. 2009. Common Trends in Software Fault and Failure Data. *IEEE Trans. Softw. Eng.* 35, 4 (2009), 484–496.

[19] Brian Henderson-Sellers. 1996. *Object-oriented Metrics: Measures of Complexity.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] I. Herraiz, G. Robles, J. M. Gonzalez-Barahona, A. Capiluppi, and J. F. Ramil. 2006. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *CSMR'06.* 206–213.

[21] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485 – 501.

[22] Maurice George Kendall. 1975. *Rank correlation methods.* Charless Griffin, London.

[23] Stefan Koch. 2007. Software evolution in open source projects—a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice* 19, 6 (2007), 361–382.

[24] Y. Lee, J. Yang, and K. H. Chang. 2007. Metrics and Evolution in Open Source Software. In *QSIC 2007.* 191–197.

[25] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang. 1995. Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow. In *ICSQ.* Maribor, Slovenia, 81–90.

[26] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology.* Springer, 108–124.

[27] Wei Li and Sallie Henry. 1993. Object-oriented metrics that predict maintainability. *Journal of systems and software* 23, 2 (1993), 111–122.

[28] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[29] T. Mens, J. Fernandez-Ramil, J. Fernandez-Ramil, and S. Degrandsart. 2008. The evolution of Eclipse. In *ICSM.* 386–395.

[30] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. 2005. Challenges in software evolution. In *IWPSE'05.* 13–22.

[31] Jeremy Miles. 2014. *R Squared, Adjusted R Squared.* American Cancer Society.

[32] P.A. Morettin and C.M. de Castro Toloi. 2006. *Time Serie Analysis.* Edgard Blucher. (In portuguese).

[33] Manfred Mudelsee. 2013. *Climate time series analysis. Classical Statistical and Bootstrap Methods.* Springer.

[34] Glenford J Myers. 1975. *Reliable Software Through Composite Design.* Petrocelli/Charter.

[35] Raphael Pereira Oliveira, Alcemir Rodrigues Santos, Eduardo Santana de Almeida, and Gecynalda Soares da Silva Gomes. 2017. Evaluating Lehman's Laws of software evolution within software product lines industrial projects. *Journal of Systems and Software* 131 (2017), 347–365.

[36] Bihrat Önöz and Mehmetcik Bayazit. 2012. Block bootstrap for Mann–Kendall trend test of serially dependent data. *Hydrological Processes* 26, 23 (2012), 3552–3560.

[37] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. 2004. Where the Bugs Are. In *ISSTA.* New York, NY, USA, 86–96.

[38] Uzma Raja, David P. Hale, and Joanne E. Hale. 2009. Modeling software evolution defects: a time series approach. *Journal of Software Maintenance and Evolution: Research and Practice* 21, 1 (2009), 49–71.

[39] Markus Lumpe Rajesh Vasa and Allan Jones. 2010. Helix - Software Evolution Data Set. http://www.ict.swin.edu.au/research/projects/helix.

[40] Ian Sommerville. 2012. *Software Engineering* (9th ed.). Pearson.

[41] Cecilia Svensson, W Zbigniew Kundzewicz, and Thomas Maurer. 2005. Trend detection in river flow series: 2. Flood and low-flow index series/Détection de tendance dans des séries de débit fluvial: 2. Séries d'indices de crue et d'étiage. *Hydrological Sciences Journal* 50, 5 (2005).

[42] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, APSEC (Ed.). IEEE, 336–345.

[43] R. P. F. Trindade, T. S. Orfanó, K. A. M. Ferreira, and E. F. Wanner. 2017. The Dance of Classes - A Stochastic Model for Software Structure Evolution. In *WETSoM.* 22–28.

[44] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering.* Springer.

[45] G. Xie, J. Chen, and I. Neamtiu. 2009. Towards a better understanding of software evolution: An empirical study on open source software. In *ICSM.* 51–60.

[46] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware Just-in-time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models. In *FSE.* ACM, New York, NY, USA, 157–168.

[47] Abdelhak M Zoubir and D Robert Iskander. 2004. *Bootstrap techniques for signal processing.* Cambridge University Press.