Received: 24 October 2022 | Revised: 12 December 2023 | Accepted: 25 December 2023

DOI: 10.1002/spe.3310

RESEARCH ARTICLE

WILEY

Evolution of internal dimensions in object-oriented software–A time series based approach

Bruno L. Sousa[1] | Mariza A. S. Bigonha[1] | Kecia A. M. Ferreira[2] | Glaura C. Franco[3]

[1]Computer Science Department, Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

[2]Department of Computing, Federal Center for Technological Education of Minas Gerais (CEFET-MG), Belo Horizonte, Brazil

[3]Department of Statistics, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil

Correspondence
Bruno L. Sousa, Computer Science Department, Federal University of Minas Gerais (UFMG), 31270-901, Belo Horizonte-MG, Brazil.
Email: bruno.luan.sousa@dcc.ufmg.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico; Fundação de Amparo à Pesquisa do Estado de Minas Gerais; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

Summary
Software evolution is the process of adapting, maintaining, and updating a software system. This process concentrates the most significant part of the software costs. Many works have studied software evolution and found relevant insights, such as Lehman's laws. However, there is a gap in how software systems evolve from an internal dimensions point of view. For instance, the literature has indicated how systems grow, for example, linearly, sub-linearly, super-linearly, or following the Pareto distribution. However, a well-defined pattern of how this phenomenon occurs has not been established. This work aims to define a novel method to analyze and predict software evolution. We based our strategy on time series analysis, linear regression techniques, and trend tests. In this study, we applied the proposed model to investigate how the internal structure of object-oriented software systems evolves in terms of four dimensions: coupling, inheritance hierarchy, cohesion, and class size. Applying the proposed method, we identify the functions that better explain how the analyzed dimensions evolve. Besides, we investigate how the relationship between dimension metrics behave over the systems' evolution and the set of classes existing in the systems that affect the evolution of these dimensions. We mined and analyzed data from 46 Java-based open-source projects. We used eight software metrics regarding the dimensions analyzed in this study. The main results of this study reveal ten software evolution properties, among them: coupling, cohesion, and inheritance evolve linearly; a relevant percentage of classes contributes to coupling and size evolution; a small percentage of classes contributes to cohesion evolution; there is no relation between the software internal dimensions' evolution. The results also indicate that our method can accurately predict how the software system will evolve in short-term and long-term predictions.

KEYWORDS
class size, cohesion, coupling, inheritance, software evolution, software metrics, time series

Abbreviations: ACF, autocorrelation; ARIMA, autoregressive moving average models; CAM, cohesion among methods; CBO, coupling between objects; COMETS, code metrics time series; DCM, dynamic coupling metric; DIT, depth of inheritance tree; LCOM, lack of cohesion; LOC, lines of code; NCP, necessary coupling; NOA, number of attributes; NOC, number of children; NOM, number of methods; PACF, partial autocorrelation; PMSE, predicted mean square error; QoS, quality of service; RFC, responses for a class; RQ, research question; TCC, tight class cohesion; UNCP, unnecessary coupling.

1 | INTRODUCTION

Software evolution involves developing, maintaining, and updating software systems for various reasons.[1] If a software system does not evolve, it risks losing market share to competitors.[2] Nevertheless, software maintenance is a challenging, complex, and time-consuming task. During a software life cycle, the internal software structure usually suffers several changes to accommodate the modifications and meet the user's demands, resulting in high costs. The literature has indicated that the maintenance phase costs comprise 85% to 90% of the total expenses that an organization spends with software.[3,4] Thus, it is essential to understand how the systems' internal structure evolves to adopt strategies to control and reduce software costs and efforts.

Understanding how software systems evolve has been a research subject in Software Engineering for decades. As a starting point, Lehman et al.[2] carried out empirical studies to understand the software evolution characteristics and provide empirical evidence about how software evolution occurs. For this purpose, they studied the evolutionary nature of an effective system and summarized their findings in eight laws known as Lehman's laws. Table 1 presents the eight software evolution laws proposed by Lehman et al.[2]

Lehman's laws are one of the landmarks of software evolution and have inspired the community to investigate this topic. Many studies in the literature have investigated software evolution aiming to check and validate the presence of Lehman's laws in software development contexts, such as open-source software,[5–10] mobile applications,[11–13] proprietary software,[14] and C library.[15] Other researches have aimed to characterize software evolution under some software dimensions,[16–29] trying to understand the real impact of some dimensions on some factors in the software, such as the emergence of faults, maintainability, and change,[30–36] and detailed how-to state these dimensions in the software.[37]

However, the studies on the evolution of software's internal dimensions have diverged in the results and have yet to reach a clear and precise conclusion. For instance, there has yet to be a consensus on how the size of a software system evolves. Some findings indicate that this dimension grows linearly,[20] while others indicate that this growth is super-linear,[16,21,25,28] sub-linear,[19,22,23] or even follows a Pareto distribution.[24] Hence, there needs to be more current knowledge of how software systems evolve. This work aims to contribute specifically to comprehending how software systems evolve from the perspective of internal dimensions.

Besides, the time series approach has often been used in software engineering mainly for extracting and proposing prediction models. Some application scenarios as defect,[38–43] changes,[44] clones,[45] size,[46] complexity,[46] and quality of service (QoS)[47] predictions are some examples of substantial applications of time series analysis in software engineering.

TABLE 1 Lehman's laws of software evolution.

No	Name	Description
I	Continuing Change	Systems must be continually adapted; else, they become progressively less satisfactory.
II	Increasing Complexity	The complexity of a system increases over its evolution unless work is done to maintain or reduce it.
III	Self Regulation	The system evolution process is self-regulating with a distribution of product and process measures close to normal.
IV	Conservation of Organizational Stability	The average effective global activity rate in an evolving system is invariant over the product's lifetime.
V	Conservation of Familiarity	As an E-type system evolves, all associated with it; developers, sales personnel, and users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes mastery; hence, the average incremental growth remains invariant as the system evolves.
VI	Continuing Growth	The functional content of the software systems must be continually increased to maintain user satisfaction over their lifetime.
VII	Declining Quality	The quality of the systems will decline unless they are through rigorous maintenance and adaptation to operational environment changes.
VIII	Feedback System	The evolution processes constitute multi-level, multi-loop, multi-agent feedback systems, and they must be treated this way to achieve significant improvement over any good base.

Source: Adapted from Lehman et al.[2]

We defined a novel method to analyze and predict software evolution. Aiming to show the practical application of our software evolution method and study how object-oriented software systems evolve from the point of view of internal dimensions, we carried out an empirical study considering four characteristics: *coupling*, *inheritance hierarchy*, *cohesion*, and *class size*. We based our analysis on these four dimensions because they are relevant software architecture characteristics that have been little explored and studied in the literature. Class size is the only characteristic that has been further investigated in the literature. However, we have little information about how coupling, inheritance hierarchy, and cohesion have evolved inside the architecture of software systems.

Our study mined and analyzed data from 46 Java-based open-source projects. We used eight software metrics regarding the dimensions analyzed in this study: fan-in and fan-out for coupling; DIT (Depth of Inheritance Tree) and NOC (Number of Children) for characterizing inheritance hierarchy; NOA (Number of Attributes) and NOM (Number of Methods) for class size; and LCOM (Lack of Cohesion) and TCC (Tight Class Cohesion) for cohesion. Fan-in indicates the number of references made to a given class by other classes, while fan-out reflects the number of calls made by a given type to other classes.[4,5] NOA and NOM are, respectively, the number of attributes and methods of class.[48] DIT indicates a class's position in its inheritance hierarchy, and NOC is the number of immediate subclasses of a given class.[49] LCOM measures the lack of cohesion between methods of class,[49] and TCC indicates the cohesion of a class via direct connections between visible methods.[50–52] The dataset used in this work is publicly available.[53] We did not consider data of LOC (Lines of Code) to measure class size because many previous studies have analyzed software evolution with this metric already.[16–18,20–25,28,29,54]

The main contributions of this work are:

1. A novel method to analyze and predict software evolution based on time series analysis (Section 4). The strategy consists of two phases. The first phase uses linear regression to model the data's evolution pattern and identify the mode type that better represents their behavior. The second one applies trend tests in the time series to analyze the classes' evolution, increasing or decreasing over time.
2. A set of ten properties of object-oriented software evolution regarding coupling, inheritance, cohesion, and class size. (Section 7).

We organized the remaining of this paper as follows. Section 2 provides a background on the software metrics considered in this study and explains the research questions investigated in this work. Section 3 describes the construction of the dataset used in this work. Section 4 presents our method for software evolution data analysis. Section 5 reports the empirical analysis of software evolution by applying the proposed method. Section 6 presents the results of the software evolution prediction using our method. Section 7 shows the evolution properties we depicted based on the results of our study. Section 8 discusses some practical implications study. Section 9 discusses the threats to validity. Section 10 presents related works. Section 11 concludes this paper and indicates future works.

2 | RESEARCH QUESTIONS

This section explains the research questions investigated in this work and describes the software metrics we used to measure coupling, size, inheritance hierarchy, and cohesion. We considered fan-in and fan-out to represent coupling, NOA (Number of Attributes), and NOM (Number of Methods) to measure size. At the same time, DIT (Depth of Inheritance Tree) and NOC (Number of Children) represent inheritance hierarchy, and LCOM (Lack of Cohesion) and TCC (Tight Class Cohesion) characterize cohesion.

Fan-in indicates the number of references made to a given class by other classes, while fan-out reflects the number of calls made by a given type to other classes.[4,5] There are many static and dynamic object-oriented software metrics for coupling, such as Yacoub's metrics,[55] Arisholm's metrics,[56] Mitchell's metrics,[57–60] DCM (Dynamic Coupling Metric),[61–63] and Gupta's metrics.[64] We decided to use fan-in and fan-out because they consider the method invocations and class attributes as coupling, provide measures at the class level, and analyze the coupling in both input and output aspects.

DIT indicates a class's position in its inheritance hierarchy, and NOC is the number of immediate subclasses of a given class.[49] NOA and NOM are size metrics, and they refer to the number of attributes and methods of a class, respectively.[48] We chose to use these metrics to measure inheritance hierarchy and size because they are well-known in the literature. Consequently, many tools are available to support their collection. We did not consider LOC (Lines of Code) to measure size because many previous studies analyzed this metric already.[16–18,20–25,28,29,54] Moreover, using NOA and NOM to analyze the size evolution in object-oriented software allows us to provide an evolutionary view of this dimension from the data and services perspective provided by the classes. LCOM and TCC are cohesion metrics. LCOM measures the lack of cohesion between methods of a class by using the notion of similarity degree of methods.[49] The range of values LCOM produces varies in the [0, 1] interval. The higher its value in a class, closer to 1, the less cohesion degree of this class. TCC also measures the cohesion of a class, however, by considering the degree of connectivity between visible methods in a class.[50–52] Similar to LCOM, TCC also produces a value between 0 and 1, however, in contrast with LCOM, the higher the TCC value in a class, the more cohesion degree of the class. We chose these two metrics because there has yet to be a consensus for measuring cohesion. Therefore, considering only one metric could bring biases to the study. LCOM and TCC have been used by other empirical studies on software metrics and are implemented by software metrics tools available in the literature. Hence, these two metrics could provide a proper analysis of cohesion evolution.[65]

The empirical analysis described in this paper aims to study the evolution of the software dimensions in open-source systems and extract properties that characterize their behavior over the software life cycle. To guide our investigation, we defined four research questions as follows.

RQ1. *Which model better describes the evolution pattern of the software systems' dimensions?*

This research question analyzes how the software dimensions evolve and identifies the patterns that better describe their behavior. With this research question, we investigate if there is a model to describe the evolution of the analyzed dimensions. We applied the behavior analysis phase of our method, described in Section 4.1, to the global time series of the software dimensions metrics to model and extract the best evolution pattern that represents them. We carried out this analysis for each dimension studied in this work. The motivation for answering this research question is to understand how the internal attributes evolve and identify the type of model that better describes the evolution of these internal characteristics.

RQ2. *How does the relation between dimension metrics behave throughout the evolution of software systems?*

This research question investigates coupling and size. In the case of coupling, we analyzed the evolution of the relationship between fan-in and fan-out. This analysis investigates how the growth of the incoming and the outgoing couplings of the classes evolve relatedly. For class size, we analyzed the relationship between NOA and NOM. This analysis investigates if the number of methods and attributes of the classes evolves in a related pattern.

Regarding inheritance, we compared the evolution of DIT and NOC from the classes of `Eclipse JDT Core` to detail preliminarily how the inheritance hierarchy evolves in this software system. We did not perform a similar analysis for cohesion because the relation between its respective metrics does not provide a direct and well-defined interpretation. We used two metrics for cohesion, LCOM and TCC, to measure a single aspect, that is, the internal cohesion of a class. The motivation for answering this research question is to investigate how the relation between related internal attributes evolves.

RQ3. *Which proportion of classes within the software system affects the growth and the decrease of the dimensions?*

This research question aims to identify the percentage of classes in a software system responsible for increasing or decreasing the internal dimensions over the software evolution. From the point of view of Software Engineering, this research question investigates the portion of classes from a system that impacts the growth of a dimension inside its structure. Knowing this portion allows researchers and practitioners to focus only on the points that contribute to the evolution of the internal dimensions inside the systems. To answer this research question, we applied the trend analysis, described in Section 4.2 as the second phase of our method, to the original time series of each class of the analyzed software systems to map the ones with a growth trend and the ones with a decreasing trend.

RQ4. *How well can our time series-based approach predict software evolution?*

This research question aims to evaluate the accuracy of our time series-based approach to predict software evolution. The motivation of this research question is to identify if the characterization models extracted in RQ1 are efficient for forecasting. For this purpose, we divided our dataset into two parts: training data and test data. With the training data, we generated the models. Predictions for future values were computed for the test data, evaluating the performance for short-term and long-term forecasts. We extracted the errors obtained for the models in each type with the test data: short-term and long-term forecasts. We compared the forecasts for each model identified using the predicted mean square error (PMSE).

3 | DATASET

We identified four datasets with software metrics: D'Ambros dataset,[66] Helix,[67] Qualitas Corpus,[68] and COMETS.[69] However, Qualitas Corpus does not provide a time series, and Helix does not include the data of all metrics we analyzed in this study. D'Ambros' dataset provides time series of coupling, cohesion, inheritance, and size metrics, comprising 90 and 99 observations for five systems. COMETS provides metrics time series comprising more observations and systems than D'Ambros' dataset. However, both D'Ambros' dataset and COMETS are outdated. Then, we decided to create an updated dataset comprising more systems, observations, and metrics. This section describes this dataset.

Although we applied eight software metrics in this study, the dataset contains time series from 46 software metrics extracted from open-source software systems available on GitHub. This section describes the steps we followed to create the dataset. Section 3.1 reports how we selected the subject projects. Section 3.2 describes how we made the releases of the systems. Section 3.3 describes how we extracted the software metrics from each release generated for the software systems. Section 3.4 describes the time series generation process. Section 3.5 presents an overview of the dataset.

3.1 | Selecting the subject systems

The first criterion to select the software systems was to include the software systems considered in COMETS dataset.[69] COMETS is a comprehensive time series dataset considering the number of systems and software metrics. It contains time series of ten Java-based software systems and has been used for studies on software evolution.[35,70] COMETS was created in 2010, that is, before GIT became the leading software repository. With the popularity of GIT, many projects considered by COMETS were migrated to GIT. Therefore, despite its coverage, COMETS is outdated. Considering the relevance of this dataset, we decided to include the software systems existing in COMETS so that their information may be updated and extended with information about other software metrics. However, `TV-Browser`, considered in COMETS, was discontinued in 2013. We omitted TV-Browser from our dataset because we are interested in systems the community has continuously maintained.

As we aimed to analyze a more extensive set of systems, we defined the following criteria to select them:

1. Programming Language. We considered Java-based software systems. We chose this programming language for the following reasons: it is the same as the COMETS dataset. Java is one of the most popular programming languages, and the software metrics tools usually collect metrics from Java programs.
2. Popularity. We considered the number of stars of the projects on GitHub to characterize their popularity. The higher the number of stars, the higher its popularity. Therefore, we considered the software systems with the highest number of stars.
3. Activity. The system must have at least 5000 commits. We used this threshold because 5000 is the lowest number of commits the software systems from COMETS have in GitHub.
4. Lifetime. As the dataset will be applied in a study on software evolution, we defined that the dataset's systems need at least five years.
5. Not be deprecated. The year of the last commit of the project should be 2020. This criterion avoids selecting software systems that the developer community has abandoned.

We implemented a Python script using a REST API from GitHub to automate the selection process and applied the defined criteria to select the systems. We obtained 37 software systems. Therefore, in total, our dataset comprises 46 software systems.

3.2 | Extracting the software systems' releases

We conducted a process to extract the systems' source code a version. We based on GitHub's information about the software systems' whole lifetime. Moreover, we also defined an interval to delineate a release. We formalized a release as source code information of software regarding bi-weeks, that is, 14 days.

To automate the data extraction, we implemented a script in Python. We used Python because it works efficiently with a large quantity of data and provides a library that deals directly with the REST API provided by GitHub. The library we used is named PyGitHub.[*] Figure 1 summarizes the sequence of steps that our script follows.

[*]https://pygithub.readthedocs.io/en/latest/index.html.

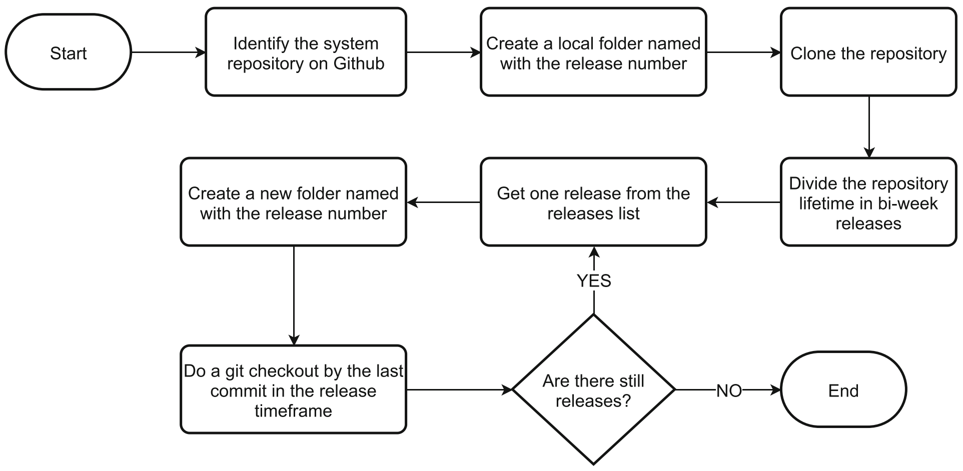

Start
Identify the system repository on Github
Create a local folder named with the release number
Clone the repository
Divide the repository lifetime in bi-week releases
Get one release from the releases list
Create a new folder named with the release number
YES
Do a git checkout by the last commit in the release timeframe
Are there still releases?
NO
End

FIGURE 1 Process of the extraction of the systems' releases.

We defined the following approach to get the released source code of the systems. We identified the repository of a particular system on GitHub and provided this repository's complete name for our approach. It is essential to highlight that the repository's full name is always composed of a user's name followed by a bar (/) and the system repository name. For instance, if we want to access the `Eclipse JDT CORE` repository, we need to inform "eclipse/eclipse.jdt.core" to the approach. Observe that "eclipse" is the owner user's name of this repository, and "eclipse.jdt.core" is the name of the system repository we want to access.

Our script created a folder using the project's name and cloned the repository inside its respective folder. With the support of PyGitHub, our script analyzed the system's information and extracted its release list, considering the time frame of the project's existence on GitHub. The project's release list is only a sequence of bi-week periods indicating each release's beginning and end. The script downloaded the source code of each release inside the project's folder, considering its release list, created a sub-folder for each release, and stored the respective source code inside this sub-folder.

3.3 | Collecting metrics' values

In this part of our methodology, we computed static metrics from the releases of the 46 software systems in our dataset. We collected 46 software metrics that characterize several aspects of the software, such as size, cohesion, inheritance hierarchy, coupling, and complexity. To compute the set of metrics, we used a tool named CK TOOL. CK TOOL is an open-source tool hosted on GitHub that computes a broad set of code metrics for Java projects at class-level, method-level, and variable-level.[71] We decided to use CK TOOL because it is easy to install, has good documentation, and is a complete tool regarding the number of metrics supported to measure Java software systems. Although CK TOOL allows extracting metrics at the method-level and variable level, we decided to compute only the class-level metrics for our dataset, which already consists of information about the software systems.

We implemented a script in the Shell script to automate the process of collecting metrics. This script calls the CK TOOL for each release of a software system, passing its source code directory. The tool exports the calculated values in *CSV* files for each release of the software systems.

3.4 | Generating time series

This part generates the time series of the software metrics values extracted from each software system's release. For this purpose, we considered the same pattern defined in References 38,69. This pattern consists of defining *CSV* files for each metric *M* and system *S*, where the lines represent the classes of *S*, the columns represent the versions, and each cell *(c,r)* consists of the metric value of class *c* in the release *r*. Then, each *CSV* contains the time series of classes regarding a given metric in a specific system. We considered only classes that refer to the systems' core functionality to extract the time series, like COMETS. Then, we discarded test classes because they do not have this characteristic and may statistically invalidate the information provided by this dataset for future prediction studies. We filtered the classes considering their directory when generating their time series to remove them. We did not generate time series for the classes kept in a directory that started with "test" as part of its complete name.

Aiming to automate the process of time series generation, we created a Python script that reads the metrics' values from the files exported by CK TOOL and reorganizes the values considering the pattern described above. At the end of this process, the script exports all metrics time series in *CSV* files.

3.5 | Dataset

Table 2 presents the 46 open-source Java systems that compose our time series dataset. It indicates the name of the systems, the repository's complete name where they are kept on GitHub, the number of releases extracted from them, and the timeframe considered in their versioning. We made this dataset publicly available and provided additional information on our supplementary Web site.[53]

4 | SOFTWARE EVOLUTION DATA ANALYSIS

The method we propose to analyze software evolution comprises two phases: behavior analysis, described in Section 4.1, and trend analysis, described in Section 4.2. We considered a significance level of 5% in all the statistical analyses.

4.1 | Behavior analysis

The Behavior Analysis phase is a process that aims to model the evolution of the time series to (i) characterize and describe its behavior or (ii) extract a prediction model for predicting future values regarding a given time series. Figure 2 shows the steps of this phase.

We describe the steps of the behavior analysis as follows.

Step 1: Time series normalization. We considered only metrics at the class level. This step normalizes the time series of classes and extracts a global serial measure for each system. We evaluated the global measure of the systems by taking the arithmetic average of their metric values. We have decided to use the arithmetic average because it gives us a real sense of how the coupling and size evolve since it uses weighting by the number of classes. Furthermore, considering a standardized form, we extract the global measure of all metrics analyzed in this study.

Step 2: Dataset preparation. This step divides the dataset into two subsets: training and test. The training data are used to model the behavior of the analyzed time series. Then, the best models are applied to the test data to evaluate the accuracy of the extracted model in terms of prediction. The training subset comprises 80% of the observations belonging to the time series. The test subset includes 20% of the observations and is used to assess the forecasting performance of the models.

Step 3: Application of linear regression method. This step involves applying the linear regression methods[72] to model the global time series. Other studies have used different approaches, such as autoregressive moving average models (ARIMA), to model the evolution of software metrics.[39,45,46,73] The ARIMA technique requires that the time series observations be collected over a well-defined time scale, such as days, months, or years.[74] In contrast, linear regression does not require the time series observations to be equally spaced over the total period. We chose the linear regression approach, which is more flexible and appropriate to our data. We modeled the metrics of coupling, cohesion, inheritance hierarchy, and size using the following types of models: (i) linear; (ii) quadratic (polynomial at Degree 2); (iii) cubic (polynomial at Degree 3); (iv) logarithmic at Degree 1; (v) logarithmic at Degree 2; and (vi) logarithmic at Degree 3. We considered all these models to identify which better describes the analyzed metrics' evolution patterns.

Step 4: Intervention analysis. Time series may be frequently affected by external factors or events. These factors may change the evolutionary behavior of an analyzed phenomenon or even affect its prediction. In the software context, refactoring and restructuring are examples of events that may impact the behavior of a time series. A system usually undergoes several modifications and refactoring processes over its lifetime that may change its time series pattern over its evolution. To treat this characteristic, we used *intervention analysis*,[75] a technique that evaluates and measures the effects

TABLE 2 Overview of the software systems included in the dataset.

ID	System name	Repository on GitHub	# of Versions	Timeframe
1	Alluxio	Alluxio/alluxio	64	2018-04-24 – 2020-12-08
2	Antlr4	antlr/antlr4	264	2010-01-28 – 2020-11-30
3	Arduino	arduino/Arduino	372	2005-08-25 – 2020-12-03
4	Bazel	bazelbuild/bazel	141	2015-02-25 – 2020-12-09
5	Bisq	bisq-network/bisq	162	2014-04-11 – 2020-12-04
6	Buck	facebook/buck	184	2013-04-18 – 2020-11-06
7	CAS	apereo/cas	252	2010-07-22 – 2020-11-25
8	CoreNLP	stanfordnlp/CoreNLP	180	2013-06-27 – 2020-11-16
9	Dbeaver	dbeaver/dbeaver	199	2012-10-03 – 2020-12-04
10	Dropwizard	dropwizard/dropwizard	223	2011-10-07 – 2020-12-02
11	Druid	alibaba/druid	232	2011-05-11 – 2020-11-18
12	Eclipse JDT Core	eclipse/eclipse.jdt.core	473	2001-06-05 – 2020-11-06
13	Eclipse PDE UI	eclipse/eclipse.pde.ui	474	2001-05-24 – 2020-11-09
14	Elasticsearch	elastic/elasticsearch	262	2010-02-08 – 2020-11-11
15	Equinox Framework	eclipse/rt.equinox.framework	414	2003-11-25 – 2020-11-24
16	FrameworkBenchmarks	TechEmpower/FrameworkBenchmarks	187	2013-03-22 – 2020-11-24
17	Gocd	gocd/gocd	162	2014-04-12 – 2020-12-05
18	Graylog	Graylog2/graylog2-server	252	2010-07-31 – 2020-12-04
19	Guava	google/guava	273	2009-09-01 – 2020-11-16
20	Hibernate Orm	hibernate/hibernate-orm	326	2007-06-29 – 2020-11-16
21	J2ObjC	google/j2objc	201	2012-09-05 – 2020-12-06
22	Jabref	JabRef/jabref	418	2003-10-14 – 2020-12-12
23	Jenkins	jenkinsci/jenkins	342	2006-11-05 – 2020-11-20
24	Jitsi	jitsi/jitsi	371	2005-07-21 – 2020-10-14
25	JMeter	apache/jmeter	86	2017-05-26 – 2020-12-05
26	JUnit 5	junit-team/junit5	125	2015-10-17 – 2020-12-03
27	K-9 Mail	k9mail/k-9	293	2008-10-28 – 2020-11-08
28	Kafka	apache/kafka	227	2011-08-01 – 2020-11-25
29	LanguageTool	languagetool-org/languagetool	73	2017-12-16 – 2020-12-14
30	Lucene	apache/lucene-solr	259	2010-03-28 – 2020-11-14
31	MinecraftForge	MinecraftForge/MinecraftForge	229	2011-07-12 – 2020-12-05
32	Neo4j	neo4j/neo4j	329	2007-05-24 – 2020-11-25
33	Netty	netty/netty	299	2008-08-08 – 2020-11-17
34	OpenRefine	OpenRefine/OpenRefine	258	2010-04-26 – 2020-11-28
35	OrientDB	orientechnologies/orientdb	260	2010-03-29 – 2020-11-30
36	Pentaho Kettle	pentaho/pentaho-kettle	329	2007-05-16 – 2020-11-17
37	Pentaho Platform	pentaho/pentaho-platform	223	2011-09-21 – 2020-11-16
38	Pinpoint	pinpoint-apm/pinpoint	153	2014-08-23 – 2020-12-03
39	PMD	pmd/pmd	449	2002-06-21 – 2020-11-27
40	Realm Java	realm/realm-java	210	2012-04-20 – 2020-12-03
41	RxJava	ReactiveX/RxJava	192	2012-12-28 – 2020-11-15
42	Spring Boot	spring-projects/spring-boot	185	2013-04-19 – 2020-11-22
43	Spring Framework	spring-projects/spring-framework	294	2008-10-23 – 2020-11-18
44	Spring Security	spring-projects/spring-security	407	2004-03-16 – 2020-12-01
45	Tomcat	apache/tomcat	358	2006-03-27 – 2020-12-07
46	Tutorials	eugenp/tutorials	184	2013-04-29 – 2020-11-17

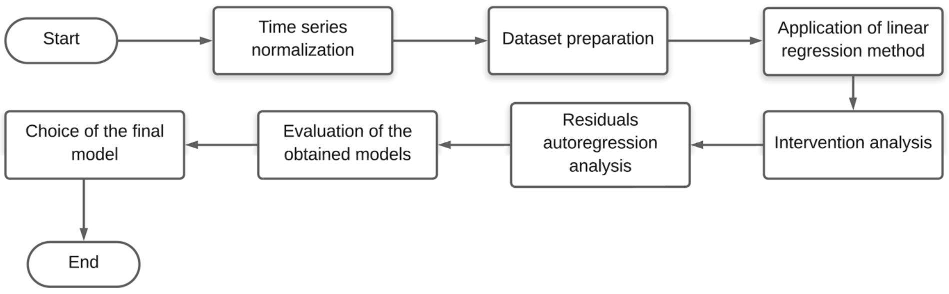

Start
Time series normalization
Dataset preparation
Application of linear regression method
Intervention analysis
Residuals autoregression analysis
Evaluation of the obtained models
Choice of the final model
End

FIGURE 2 Steps of the behavior analysis phase.

these external factors cause in the time series. This technique generally involves dummy variables to point out where the intervention occurred and indicate how this occurrence will impact the following time series values. Hence, this step involves checking the change points that may influence the time series behavior and conducting an intervention analysis to adjust the model to the new pattern. Therefore, the intervention analysis allows us to improve the representation quality of the models.

Step 5: Residuals autoregression analysis. Regression methods require that the assumption of independence be satisfied to ensure the validity of the models.[76] Using linear regression to model time series may lead to autocorrelated error terms. *Autocorrelation* consists of a serial dependence between their values.[77] If we do not treat this autocorrelation, the coefficients' estimates and their standard errors may be wrong, and the model will not correctly represent the time series.[76] Step 5 evaluates the models' errors and incorporates the autocorrelation by modeling them via autoregressive models. *Autoregression* is a process that uses observations from previous time steps to model the value at the next time.[77] After modeling the residuals, we incorporated the autoregressive error coefficient in its respective model.

Step 6: Evaluation of the obtained models. To assess the models' adequacy, we computed their adjusted determination coefficient ($\overline{R}^2$), a metric extracted from the linear regression analysis, which considers the number of parameters introduced in the model and penalizes the inclusion of less critical parameters. $\overline{R}^2$ measures the adjustment of a model to the data, allowing us to understand to which extent the model explains the variability of analyzed data.[78]

Step 7: Choice of the final model. It compares the best models obtained for the systems time series and selects the type that better describes the metrics' behavior. Using intervention and autoregression analysis to improve the models' fit may result in $\overline{R}^2$ values very high and close to each other. Then, we defined an evaluation protocol to compare the values and choose the best model. Our protocol comprises three criteria, which evaluate a different aspect of the model. We describe these criteria as follows.

1. **Relevance.** We selected the generated models with values of $\overline{R}^2$ higher than or equal to 80% since this percentage already defines models with good adjustment.
2. **Coverage.** We selected models covering the most significant number of systems.
3. **Simplicity.** We selected the simplest model, considering the following order: (i) linear, (ii) quadratic, (iii) cubic, (iv) logarithmic at Degree 1, (v) logarithmic at Degree 2, and (vi) logarithmic at Degree 3.

4.2 | Trend analysis

Trend analysis is the second phase of our method and comprises seven steps.

Step 1: Data organization. When a given class is absent in a particular system observation, we set its metrics to −1. In the context of our analysis, −1 values may bias the results. Hence, we removed them from the time series and reorganized the observations, considering only the valid values.

Step 2: Removal of ghost classes. Changes in a system may result in *ghost classes*. This phenomenon consists of breaks that divide the time series of a class into several sub-series. Figure 3 illustrates this phenomenon. In this example, the class `LayoutTest` was introduced in the system in the eighth period. This class remained there for some periods, and at a specific moment, it was removed from the system. However, between the 200th and 300th periods it was reintroduced in the system. This class being removed and reintroduced in the system characterizes a ghost class occurrence. Although it is essential to recognize the legitimacy of ghost classes as a software evolution pattern, it is also important to highlight that considering them may introduce bias in our analysis. When a class does not appear in some periods and appears in the subsequent release, as it happened between the 200^{th} and 300^{th} periods in Figure 3, it generates a break in the time series. Trend tests work only with continuous time series, that is, without breaks. Applying trend tests in time series with breaks makes the test fail and trend analysis unfeasible in these cases. Due to this reason, in this step, we disregard time series with this characteristic from our analysis.

Step 3: Application of trend tests. This step involves applying trend tests in the time series to identify whether it has a growth or a decreasing trend. We used three different statistical tests to evaluate the presence of trends in the series: (i) Mann-Kendall;[79] (ii) Cox-Stuart,[80] and (iii) Wald-Wolfowitz.[80] We chose them because they have been pointed out as applicable and efficient.[79,80] We considered the following hypotheses:

1. $\mathbf{H_0}$. There is no trend in the time series
2. $\mathbf{H_1}$. There is a trend in the time series.

The statistical tests may contain weaknesses and be prone to errors even when we choose efficient trend tests. Hence, we defined the following criteria to determine the presence of a trend: *"the time series has a trend if, and only if, the null hypothesis is rejected at least in two of the three tests"*.

It is essential to highlight that removing −1 values from a time series may substantially reduce the number of observations in some of them. Therefore, we decided to only analyze and apply the trend tests in time series with ten or more continuous observations. Times series with less than ten measures are disregarded and classified as having no trend.

Step 4: Identification of autocorrelated time series. Mann-Kendall test is sensitive to the presence of autocorrelation. When the original version of the Mann-Kendall test is carried out in an autocorrelated time series, it may generate false positives or negatives.[81] This step analyzes the time series to identify autocorrelation and avoid this problem. We designed and developed an automatic checking approach aiming to facilitate our analysis. This approach examines the plots of time series autocorrelation (ACF) and partial autocorrelation (PACF). *ACF* is a correlation of any series with its lagged values plotted along with the confidence band.[74] It describes how well a given value is related to its past observations. *PACF* consists of a plot of the partial correlation of the series with its own lagged values regressed at shorter lags. Non-stationary series were properly made stationary by taking successive differences in the original series.

Step 5: Application of the modified Mann-Kendall test. This step applies a modified Mann-Kendall test approach to deal with autocorrelation. Hamed and Rao[81] derived a theoretical relationship to calculate the original test variance for autocorrelated data. These theoretical results modified the value of the variance from the original test and proposed a modified approach that was more suitable and powerful. Therefore, we used this approach to analyze the autocorrelated time series.

Step 6: Identification of trends. We apply the criteria defined in Step 3 to identify the time series with the trend, using the Cox-Stuart, Wald-Wolfowitz, and Mann-Kendall test (or the modified Mann-Kendall test if the series has autocorrelation).

Step 7: Classification of trends. It evaluates the trend, and we analyze the time series behavior and classify them as follows.

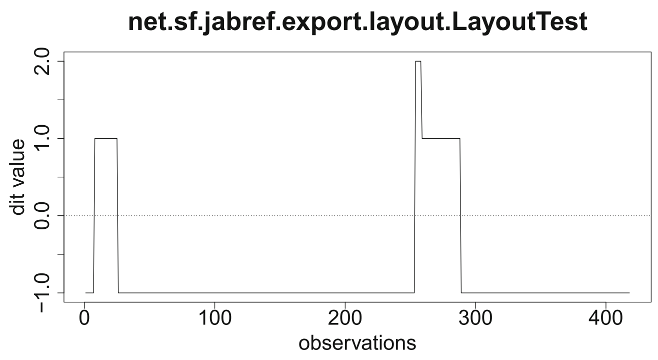


FIGURE 3 Time series of a ghost class.

1. **Upward trend.** It is a pattern whose distance between the trend line and the x-axis increases over the x-axis
2. **Downward trend.** It is a pattern whose distance between the trend line and the x-axis decreases over the x-axis
3. **Undefined trends.** They are cases that do not follow a clear pattern or whose values of the first observation are equal to the last.

5 | RESULTS

This section presents the results of our analysis by answering RQ1, RQ2, RQ3, and RQ4.

5.1 | Evolution properties at the system level

This section answers RQ1.

RQ1. *Which model better describes the evolution pattern of the software systems' dimensions?*

In this research question, we investigated how the coupling, cohesion, inheritance hierarchy, and class size dimensions evolve and identify the pattern that better describes their properties' behavior.

We applied the first phase of our method (Section 4.1) to the global time series metrics to evaluate how they evolve. As we perform a forecasting analysis in Section 6, the time series will be split into two subsets: training and test. Thus, the analysis in this section comprises the modeling using only the training data.

We analyzed the metrics' behavior before analyzing $\overline{R}^2$. We plotted the global time series of the metrics as line charts to evaluate if the coupling, cohesion, inheritance hierarchy, and class size increased or decreased over time. Due to space limitations, we did not include the time series charts in this paper. However, we made them available as supplementary material.[†]

Analyzing the charts built for each global time series, we counted the number of systems that presented growth or decrease patterns in each metric. Then, we applied the Signal test[82] to identify any difference between the proportion of increase and decrease in each system, aiming to determine which pattern the systems tend to follow over time for the analyzed metrics. We consider the following hypotheses:

1. $\mathbf{H}_0$: the proportion of growth and decrease is the same.
2. $\mathbf{H}_1$: the proportion of growth and decrease differs.

Table 3 shows the number of systems for each metric with a growth or decrease pattern. In Table 3, we also report the signal test results for each metric.

In the results exhibited in Table 3, three metrics presented p-values less than 0.05: fan-in, fan-out, and NOC. Then, we can conclude that, in most analyzed systems, the coupling, given by fan-in and fan-out, has increased. Besides, the increasing pattern of NOC means that the breadth of the inheritance hierarchies tends to increase. Also, the results shown in Table 3 reveal that, for TCC, there are more systems with a growing pattern of this metric than systems with a decreasing pattern. Although this behavior is not significant at a 5% level, it is significant at a 10% level. TCC measures the internal cohesion of a class, considering only public methods. Then, this result indicates that the classes have become more cohesive over the systems' evolution when considering the classes' public services,

Regarding DIT, LCOM, NOA, and NOM, the signal test has not pointed out a relevant difference between the number of systems with a growing and decreasing pattern from the global perspective of these metrics. Then, we cannot extract a well-defined pattern for the evolution of these metrics in terms of growth or decrease. Nevertheless, the descriptive analysis shown in Table 3 indicates that most systems have a decreasing pattern for DIT, LCOM, NOA, and NOM. This result suggests that, in most systems, the inheritance hierarchies' depth, the classes' cohesion, and the classes' size increases over time.

After identifying the software metrics' evolution patterns, we modeled their global evolution using regression techniques. We applied our evaluation protocol, Section 4.1, considering the $\overline{R}^2$ values obtained from the models to detect which model better characterizes their global evolution. Tables 4,5,6, and 7 summarize the $\overline{R}^2$ scores computed for the

[†]https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch.

TABLE 3 Number of systems whose metrics grow and decrease.

	DIT	NOC	Fan-in	Fan-out	LCOM	TCC	NOA	NOM
Growth	20	31	34	34	18	30	20	23
Decrease	26	15	12	12	28	16	26	23
p-value obtained with the Signal test	0.46	0.03	0.00	0.00	0.18	0.05	0.46	1.00

models fitted regarding analyzed metrics dimensions. The “lin.”, “quad.”, “cub.”, “log. 1”, “log. 2”, and “log.3” columns indicate the $\overline{R}^2$ values extracted from the linear, quadratic, cubic, logarithmic at degree 1, logarithmic at degree 2, and logarithmic at degree 3 models, respectively. We highlighted the $\overline{R}^2$ values in Tables 4,5,6, and 7 according to the following colors: green indicates that the values were selected in Stage 1 of our protocol; yellow indicates the models chosen in Stage 2; red indicates the models selected at Stage 3, which corresponds to the model that better characterizes the evolution of the corresponding metric in the software systems.

Analyzing the results obtained regarding DIT models in Table 4, we observe that most of the produced models have an $\overline{R}^2$ score higher than 80%. We also identify some exceptions in which the types of models could not extract any model to the time series of some systems. Therefore, we did not highlight them with green. Considering Stage 2 of our protocol, we observed that linear and logarithmic at degree 1 extracted a good fit for the analyzed time series. However, applying the criteria of Stage 3, we concluded that the linear model is the one that better explains the evolution of DIT in the analyzed software system.

The results obtained for NOC in Table 4 show that most of the generated models had $\overline{R}^2$ values higher than 80%. `Bazel`, `Bisq`, `Eclipse PDE UI`, `Elasticsearch`, `Equinox Framework`, `Framework Benchmarks`, `Gocd`, `Hibernate Orm`, `JMeter`, `Kafka`, `Lucene`, `MinecraftForge`, `Neo4j`, `Netty`, and `Pentaho Platform` were the systems for which our method did not identify any model with relevant values of $\overline{R}^2$. The logarithmic at Degree 1 was the only one selected in Stage 2 of our protocol. However, we concluded that the logarithmic at Degree 1 model is the one that better describes the growth evolution of NOC.

Table 5 shows the modeling results of fan-in and fan-out time series. We found several significant models for these two metrics in Stage 1 of our protocol. The only exceptions were the systems: `Alluxio`, `Antlr4`, `Bazel`, `CoreNLP`, `Eclipse PDE UI`, `Elasticsearch`, `Gocd`, `Jitsi`, `JMeter`, `Kafka`, `LanguageTool`, `Lucene`, `Minecraft-Forge`, `Neo4j`, `Netty`, `OpenRefine`, `Pentaho Platform`, `PMD`, `RxJava`, `Tutorials` for which we were not able to find some good models that fit the behavior of these systems in fan-in, and fan-out. Applying Stage 2 of our evaluation protocol, we selected three and one type of model for fan-in and fan-out, respectively. They are (i) linear, cubic, and logarithmic at Degree 1; (ii) linear. By applying Stage 3 of our protocol, we conclude that the linear model is the one that better describes the growth evolution of both fan-in and fan-out over the systems’ lifetime.

Table 6 exhibits the modeling LCOM and TCC time series results. The results show various relevant models obtained for these two metrics. `Alluxio`, `Bazel`, `Eclipse PDE UI`, `Elasticsearch`, `FrameworkBenchmarks`, `Gocd`, `Jitsi`, `Junit 5`, `Kafka`, `LanguageTool`, `Lucene`, `MinecraftForge`, `Netty`, `OpenRefine`, `OrientDB`, `Pentaho Platform`, `PMD`, and `Spring Boot` were the ones for which we could not find a significant model for at least one of the analyzed types for LCOM or TCC. The models we identified as non-relevant, that is, with the $\overline{R}^2$ less than 80%, were not selected at Stage 1 of our protocol. Applying the coverage criteria, we selected two types of models for LCOM and TCC, respectively. They are (i) linear and quadratic and (ii) quadratic. By applying the simplicity criteria, we conclude that the linear model is the one that better describes the evolution of LCOM. In contrast, the quadratic model is the one that represents the evolution of TCC.

Finally, Table 6 reports the results obtained for modeling NOA and NOM time series. Like the other metrics, we acquired many relevant models for NOA and NOM in Stage 1 of our protocol. We did not select in Stage 1 the cases where it was impossible to define a suitable model. Applying the coverage criteria of our protocol, we found two types, linear and logarithmic at Degree 1, for both NOA and NOM. However, considering the simplicity criteria, we conclude that the linear model is the one that better describes the evolution of NOA and NOM.

Summary of RQ1. The global coupling and the breadth of inheritance hierarchy grow over software evolution. Regarding the depth of inheritance hierarchy, cohesion, and size, we did not identify a statistical significance pattern that characterizes the evolution of these characteristics. Nevertheless, we observed that most systems have become more cohesive for this dataset. They have decreased in terms of features and data, and their inheritance hierarchy has decreased in

TABLE 4 $\overline{R}^2$ values computed from the DIT and NOC models.

	DIT						NOC					
System	**lin.**	**quad.**	**cub.**	**log. 1**	**log. 2**	**log. 3**	**lin.**	**quad.**	**cub.**	**log. 1**	**log. 2**	**log. 3**
Alluxio	81.34%	83.42%	—	81.05%	83.16%	—	88.88%	87.34%	89.40%	90.43%	87.46%	90.63%
Antlr4	97.27%	97.26%	97.33%	97.23%	97.22%	97.29%	95.11%	95.31%	95.38%	93.25%	94.16%	94.54%
Arduino	93.01%	92.71%	89.94%	92.70%	92.49%	89.59%	94.85%	94.70%	94.67%	95.84%	96.44%	96.06%
Bazel	97.70%	97.70%	97.78%	97.64%	97.64%	97.72%	96.71%	96.82%	—	96.73%	96.85%	—
Bisq	92.48%	91.91%	92.64%	93.24%	93.31%	93.36%	91.79%	90.53%	—	90.24%	90.16%	90.66%
Buck	90.88%	90.81%	91.19%	91.04%	90.98%	91.33%	83.19%	83.80%	84.15%	86.10%	86.70%	86.95%
CAS	97.23%	97.33%	97.14%	97.01%	97.12%	—	99.00%	99.03%	98.99%	98.86%	98.88%	98.89%
CoreNLP	92.50%	92.88%	93.38%	92.41%	92.80%	93.30%	94.46%	94.45%	94.51%	94.73%	94.71%	94.78%
Dbeaver	98.63%	96.97%	98.67%	98.59%	96.88%	98.62%	99.61%	99.62%	99.63%	99.57%	99.58%	99.59%
Dropwizard	82.61%	82.52%	80.24%	82.53%	82.44%	82.41%	87.59%	83.96%	87.86%	85.32%	82.05%	85.79%
Druid	90.52%	89.82%	90.49%	90.19%	89.49%	90.15%	91.19%	91.25%	90.31%	89.98%	90.01%	89.01%
Eclipse JDT Core	98.89%	98.88%	98.88%	99.05%	99.04%	99.04%	97.59%	97.59%	97.58%	97.39%	97.53%	97.53%
Eclipse PDE UI	73.37%	72.61%	73.53%	65.55%	64.50%	65.31%	64.85%	95.37%	95.42%	98.73%	98.74%	98.75%
Elasticsearch	89.36%	—	88.60%	90.05%	89.88%	—	65.41%	65.37%	67.06%	64.73%	63.75%	65.79%
Equinox Framework	—	83.20%	—	—	83.40%	—	93.70%	93.93%	—	94.40%	94.57%	—
FrameworkBenchmarks	97.55%	97.64%	97.76%	97.58%	97.67%	97.78%	98.78%	98.77%	—	98.88%	98.60%	—
Gocd	75.52%	—	—	74.92%	73.45%	—-	81.78%	82.39%	-	80.39%	80.74%	81.73%
Graylog	75.77%	75.29%	75.96%	75.62%	75.08%	75.74%	97.23%	96.02%	96.79%	96.85%	96.87%	98.99%
Guava	95.62%	95.19%	95.62%	95.76%	95.36%	95.76%	94.68%	94.80%	94.91%	94.40%	94.53%	94.63%
Hibernate Orm	92.42%	91.62%	92.48%	92.89%	92.10%	92.94%	65.46%	66.33%	63.40%	72.58%	73.13%	71.79%
J2ObjC	98.91%	98.90%	98.90%	99.01%	99.01%	99.01%	82.44%	82.69%	83.02%	84.55%	84.56%	84.78%
Jabref	98.97%	98.96%	98.98%	98.98%	98.98%	98.89%	99.35%	99.36%	99.22%	99.30%	99.30%	99.31%
Jenkins	97.24%	97.17%	97.28%	97.23%	97.16%	97.26%	98.85%	98.78%	98.87%	98.84%	98.77%	98.87%
Jitsi	45.59%	—	43.14%	42.15%	—	43.02%	80.66%	91.07%	82.78%	80.58%	90.99%	82.71%
JMeter	95.54%	95.09%	96.62%	95.42%	95.01%	96.51%	—	73.13%	74.14%	—	72.86%	73.79%
JUnit 5	98.33%	97.78%	98.32%	98.27%	97.72%	98.26%	99.28%	99.28%	99.29%	99.37%	99.37%	99.37%
K-9 Mail	92.49%	91.85%	92.70%	93.05%	92.66%	93.26%	97.31%	97.33%	96.29%	97.55%	97.59%	—
Kafka	85.80%	—	—	85.84%	—	—	82.45%	—	—	82.43%	—	—
LanguageTool	83.96%	93.82%	86.61%	83.58%	93.96%	86.27%	93.31%	93.44%	93.50%	93.24%	93.36%	93.41%
Lucene	64.83%	66.31%	68.46%	73.76%	74.12%	77.16%	60.50%	62.62%	65.20%	79.09%	79.15%	79.94%
MinecraftForge	60.19%	56.94%	55.22%	55.86%	52.75%	50.76%	72.85%	68.31%	69.88%	47.60%	48.08%	—
Neo4j	97.91%	97.93%	97.29%	97.87%	97.88%	97.29%	94.46%	94.64%	—	94.81%	94.94%	94.75%
Netty	97.37%	95.33%	96.48%	97.29%	95.36%	96.48%	85.90%	81.86%	77.41%	85.59%	81.72%	77.04%
OpenRefine	79.03%	79.20%	79.64%	77.49%	77.58%	77.98%	87.50%	88.28%	88.36%	87.40%	87.84%	87.92%
OrientDB	80.69%	79.63%	80.96%	80.83%	79.77%	81.10%	94.98%	95.35%	95.28%	94.87%	95.29%	95.22%
Pentaho Kettle	99.05%	99.07%	99.09%	99.06%	99.10%	99.10%	99.47%	99.50%	99.49%	99.50%	99.54%	99.53%
Pentaho Platform	95.84%	95.88%	95.86%	95.87%	95.91%	95.88%	93.00%	93.42%	93.43%	94.38%	94.67%	—
Pinpoint	96.52%	96.40%	96.57%	96.43%	96.42%	96.48%	98.28%	98.28%	98.30%	98.12%	98.00%	98.15%

(Continues)

TABLE 4 (Continued)

	DIT						NOC					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
PMD	97.13%	—	97.26%	97.61%	—	97.71%	89.28%	89.35%	86.67%	87.24%	87.28%	84.25%
Realm Java	95.63%	95.79%	95.72%	95.58%	95.75%	95.67%	93.74%	94.06%	93.83%	93.77%	94.12%	93.07%
RxJava	84.82%	85.49%	—	84.96%	85.83%	—	92.34%	91.88%	92.38%	88.76%	88.61%	89.24%
Spring Boot	87.28%	88.95%	85.72%	87.14%	88.83%	85.68%	90.33%	90.97%	91.20%	89.76%	90.32%	90.75%
Spring Framework	99.77%	99.77%	99.77%	99.77%	99.78%	99.78%	99.67%	99.69%	99.66%	99.66%	99.68%	99.65%
Spring Security	98.95%	98.95%	99.43%	98.99%	98.99%	99.44%	97.80%	98.08%	98.08%	97.93%	98.27%	98.26%
Tomcat	99.50%	99.51%	99.15%	99.49%	99.49%	99.14%	99.29%	98.33%	98.98%	99.24%	98.21%	98.94%
Tutorials	89.77%	92.11%	—	89.84%	92.20%	—	93.27%	93.22%	93.79%	93.45%	93.40%	93.86%

Note: (i) White: indicates models that do not have a good fit; (ii) Green: indicates models with $\overline{R}^2$ more than 80%; (iii) Yellow: indicates the type(s) of model that better modeled the most of analyzed systems for that metric; (iv) Red: indicates the models selected at Stage 3, which corresponds to the model that better characterizes the evolution of the corresponding metric in the software systems.

TABLE 5 $\overline{R}^2$ values computed from the fan-in and fan-out models.

	Fan-in						Fan-out					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	92.73%	91.43%	—	92.80%	92.88%	—	91.56%	92.55%	—	91.69%	92.64%	—
Antlr4	98.29%	98.42%	98.43%	98.53%	98.59%	98.61%	75.24%	73.77%	75.34%	74.63%	72.83%	74.73%
Arduino	94.90%	94.94%	94.94%	90.44%	90.41%	90.38%	82.64%	80.93%	81.72%	74.77%	73.36%	72.79%
Bazel	97.55%	—	96.21%	97.50%	—	96.04%	99.35%	99.34%	98.85%	99.33%	99.33%	98.81%
Bisq	98.26%	98.33%	98.40%	97.50%	97.61%	97.83%	97.04%	97.13%	96.84%	96.38%	96.53%	96.33%
Buck	94.29%	93.83%	94.22%	94.21%	94.17%	94.13%	98.99%	99.01%	99.01%	99.06%	99.08%	99.08%
CAS	99.24%	99.24%	99.26%	99.38%	99.35%	99.40%	99.77%	99.77%	99.78%	99.83%	99.83%	99.83%
CoreNLP	97.45%	97.49%	97.50%	97.48%	97.52%	97.53%	98.65%	98.72%	—	98.57%	98.66%	98.58%
Dbeaver	99.44%	99.46%	99.47%	99.43%	99.44%	99.45%	99.58%	99.59%	99.60%	99.58%	99.59%	99.60%
Dropwizard	92.80%	93.05%	98.24%	93.37%	93.56%	98.48%	94.77%	94.77%	94.88%	94.71%	94.71%	94.82%
Druid	97.63%	97.62%	97.65%	97.23%	97.22%	97.27%	97.63%	97.62%	97.65%	97.32%	97.31%	97.35%
Eclipse JDT Core	98.00%	98.28%	98.28%	98.13%	98.40%	98.39%	98.47%	98.47%	98.47%	98.74%	98.74%	98.73%
Eclipse PDE UI	63.68%	66.00%	62.99%	37.91%	42.59%	35.55%	87.19%	87.59%	86.97%	86.37%	86.84%	85.67%
Elasticsearch	84.87%	84.84%	86.18%	85.64%	85.59%	86.32%	72.25%	—	78.17%	72.14%	—	77.66%
Equinox Framework	98.73%	98.48%	98.83%	98.72%	98.43%	98.78%	96.66%	96.71%	96.49%	96.56%	96.60%	96.38%
FrameworkBenchmarks	99.40%	99.34%	99.39%	99.42%	99.42%	99.41%	99.23%	99.13%	99.23%	99.25%	99.16%	99.24%
Gocd	49.18%	49.50%	54.14%	—	58.47%	60.58%	77.22%	78.52%	79.77%	70.77%	67.25%	72.26%
Graylog	81.11%	81.02%	80.95%	80.41%	80.32%	80.26%	85.98%	86.67%	86.68%	84.94%	83.63%	83.61%
Guava	96.61%	96.61%	96.67%	96.61%	96.61%	96.68%	92.94%	92.93%	92.91%	92.92%	92.91%	92.89%
Hibernate Orm	97.77%	97.89%	97.75%	98.15%	98.44%	98.40%	85.71%	81.70%	86.63%	86.71%	82.73%	87.57%
J2ObjC	94.08%	93.97%	94.29%	92.62%	92.23%	92.91%	97.59%	97.62%	97.69%	96.43%	96.39%	96.61%
Jabref	96.99%	96.99%	97.03%	96.90%	96.90%	96.87%	98.06%	98.05%	98.20%	98.06%	98.06%	98.26%
Jenkins	95.32%	95.37%	95.44%	95.42%	95.46%	95.53%	94.10%	94.29%	94.42%	94.22%	94.41%	94.54%
Jitsi	97.63%	97.96%	97.10%	97.61%	97.96%	97.12%	97.91%	—	97.93%	97.95%	—	97.97%

(Continues)

TABLE 5 (Continued)

	Fan-in						Fan-out					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
JMeter	74.39%	76.44%	—	74.38%	76.42%	—	83.03%	87.56%	90.71%	83.39%	88.08%	90.90%
JUnit 5	98.33%	98.33%	98.38%	98.28%	98.28%	98.33%	97.69%	97.68%	97.72%	97.42%	97.41%	97.46%
K-9 Mail	98.60%	98.29%	98.29%	98.16%	98.16%	98.15%	93.65%	94.34%	93.99%	93.32%	93.67%	93.18%
Kafka	—	78.31%	78.02%	—	78.19%	77.90%	88.09%	—	90.76%	88.03%	—	90.66%
LanguageTool	75.27%	75.90%	—	75.24%	75.87%	—	71.71%	71.26%	71.70%	70.79%	70.31%	70.39%
Lucene	45.36%	—	—	37.58%	—	—	55.65%	—	48.20%	31.79%	—	—
MinecraftForge	52.39%	52.46%	48.53%	43.29%	43.54%	44.06%	51.59%	53.70%	49.97%	39.64%	40.19%	40.45%
Neo4j	77.30%	77.25%	—	77.12%	77.07%	—	88.66%	88.61%	—	88.86%	88.81%	—
Netty	75.65%	—	—	75.05%	—	—	67.52%	—	67.54%	66.23%	—	66.91%
OpenRefine	82.58%	83.44%	83.36%	85.50%	86.48%	85.88%	37.09%	40.22%	—	27.27%	—	—
OrientDB	97.58%	97.60%	97.59%	97.46%	97.49%	97.48%	98.34%	98.34%	98.31%	98.25%	98.25%	98.22%
Pentaho Kettle	95.73%	95.72%	95.72%	95.40%	95.39%	95.39%	92.89%	92.85%	92.93%	92.86%	92.82%	92.90%
Pentaho Platform	84.10%	—	84.99%	81.04%	—	79.83%	87.89%	88.46%	—	86.96%	87.52%	—
Pinpoint	96.01%	95.44%	96.21%	96.12%	95.35%	96.33%	98.74%	98.66%	98.76%	98.83%	98.78%	98.90%
PMD	90.96%	90.90%	89.96%	93.10%	93.06%	93.41%	95.93%	95.91%	—	96.43%	96.42%	96.57%
Realm Java	93.71%	94.45%	94.67%	94.05%	94.77%	95.02%	93.07%	93.33%	93.39%	93.26%	93.48%	93.56%
RxJava	79.03%	79.71%	80.63%	78.18%	78.66%	79.15%	66.30%	66.49%	66.33%	60.25%	61.45%	60.73%
Spring Boot	94.18%	94.11%	94.24%	93.74%	93.68%	93.86%	88.03%	88.06%	90.22%	87.71%	87.76%	89.91%
Spring Framework	99.50%	99.51%	99.51%	99.48%	99.48%	99.48%	99.59%	99.60%	99.61%	99.60%	99.55%	99.62%
Spring Security	88.74%	88.73%	87.86%	87.56%	87.56%	86.80%	97.89%	97.92%	97.91%	97.77%	97.79%	97.79%
Tomcat	97.53%	97.55%	97.55%	97.56%	97.58%	97.59%	97.49%	97.54%	97.53%	97.49%	97.53%	97.52%
Tutorials	94.41%	94.84%	95.31%	95.17%	95.51%	96.04%	91.50%	—	91.89%	92.14%	—	92.49%

Note: (i) White: indicates models that do not have a good fit; (ii) Green: indicates models with $\overline{R}^2$ more than 80%; (iii) Yellow: indicates the type(s) of model that better modeled the most of analyzed systems for that metric; (iv) Red: indicates the models selected at Stage 3, which corresponds to the model that better characterizes the evolution of the corresponding metric in the software systems.

TABLE 6 $\overline{R}^2$ values computed from the LCOM and TCC models.

	LCOM						TCC					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	91.83%	92.50%	—	91.94%	92.72%	—	88.64%	86.58%	88.95%	89.79%	86.55%	89.84%
Antlr4	92.29%	92.68%	92.56%	92.32%	92.78%	92.77%	89.27%	89.26%	89.05%	89.25%	89.24%	89.04%
Arduino	87.29%	86.93%	86.07%	84.91%	84.05%	82.70%	92.71%	92.89%	92.82%	89.15%	85.92%	88.84%
Bazel	97.67%	97.67%	—	97.65%	97.65%	—	94.67%	93.05%	95.05%	94.59%	93.00%	94.98%
Bisq	93.29%	93.76%	93.87%	93.28%	93.81%	93.89%	91.23%	91.25%	91.65%	90.27%	90.32%	90.85%
Buck	95.21%	95.18%	95.30%	95.12%	95.09%	95.21%	95.28%	95.35%	95.56%	95.35%	95.43%	95.64%
CAS	93.89%	93.94%	94.05%	94.33%	94.38%	94.48%	98.37%	98.38%	98.38%	98.21%	98.23%	97.78%
CoreNLP	92.78%	92.75%	92.78%	92.95%	92.93%	92.95%	95.74%	95.75%	96.04%	95.66%	95.66%	95.72%
Dbeaver	98.66%	95.05%	98.77%	98.57%	94.82%	98.68%	88.82%	89.72%	89.99%	88.70%	89.60%	89.90%
Dropwizard	98.09%	98.13%	98.18%	98.17%	98.21%	98.25%	88.56%	89.71%	89.64%	88.21%	89.23%	89.17%

(Continues)

TABLE 6 (Continued)

	LCOM						TCC					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
Druid	99.25%	99.26%	99.26%	99.32%	99.32%	99.33%	98.03%	98.02%	98.03%	98.01%	98.00%	98.01%
Eclipse JDT Core	98.73%	98.72%	98.72%	98.91%	98.90%	98.90%	98.31%	97.42%	98.34%	98.22%	97.32%	98.26%
Eclipse PDE UI	97.75%	52.14%	97.56%	96.60%	74.75%	73.13%	68.70%	68.65%	68.90%	64.58%	64.53%	64.78%
Elasticsearch	83.00%	81.22%	83.67%	77.88%	76.55%	78.27%	76.67%	—	77.21%	71.77%	61.74%	66.31%
Equinox Framework	87.85%	87.98%	87.27%	87.32%	87.42%	86.63%	95.69%	95.58%	95.75%	95.06%	94.59%	95.12%
FrameworkBenchmarks	94.05%	93.01%	93.98%	94.06%	92.98%	93.99%	—	96.04%	96.26%	—	96.08%	96.30%
Gocd	34.63%	30.03%	42.90%	42.02%	42.79%	49.39%	46.56%	44.38%	48.76%	49.22%	48.11%	54.18%
Graylog	97.74%	97.83%	97.97%	98.15%	98.24%	98.35%	93.14%	92.66%	93.16%	92.89%	89.38%	92.90%
Guava	96.44%	96.44%	95.29%	96.47%	96.46%	95.27%	96.88%	96.89%	96.94%	96.79%	96.80%	96.85%
Hibernate Orm	93.73%	93.70%	93.79%	93.49%	93.45%	93.54%	96.54%	96.23%	96.56%	96.44%	96.13%	96.46%
J2ObjC	88.30%	88.56%	88.76%	88.04%	88.30%	88.50%	98.23%	98.29%	98.32%	98.36%	98.41%	98.44%
Jabref	96.93%	96.93%	96.97%	96.86%	96.87%	96.91%	99.08%	99.09%	99.14%	99.03%	99.04%	99.08%
Jenkins	95.23%	95.24%	95.30%	95.36%	95.37%	95.42%	98.75%	98.75%	98.75%	98.74%	98.74%	98.74%
Jitsi	82.77%	80.91%	86.21%	82.80%	80.94%	86.27%	—	86.05%	—	—	86.24%	—
JMeter	83.29%	84.98%	89.66%	83.45%	85.66%	89.79%	88.39%	88.61%	87.66%	88.84%	89.03%	87.91%
JUnit 5	98.87%	98.87%	98.90%	98.63%	98.63%	98.67%	94.48%	97.45%	—	94.38%	97.32%	—
K-9 Mail	97.81%	97.13%	97.88%	98.28%	97.46%	98.35%	91.11%	94.58%	91.28%	90.56%	94.37%	90.78%
Kafka	—	—	61.31%	—	—	61.27%	95.08%	95.44%	—	95.14%	95.51%	—
LanguageTool	96.72%	97.00%	96.89%	96.80%	97.13%	—	92.93%	93.05%	—	93.56%	93.67%	—
Lucene	58.81%	—	59.66%	29.96%	—	37.69%	60.41%	61.84%	—	44.11%	46.03%	48.76%
MinecraftForge	57.91%	55.80%	55.56%	39.92%	39.06%	38.87%	41.53%	43.91%	39.80%	33.70%	36.55%	34.63%
Neo4j	98.89%	98.89%	98.88%	98.97%	98.97%	98.97%	91.15%	89.69%	91.12%	91.25%	89.83%	91.22%
Netty	67.85%	68.41%	69.47%	66.98%	67.50%	68.52%	79.36%	80.48%	79.96%	78.22%	79.35%	78.79%
OpenRefine	46.84%	46.85%	47.70%	36.71%	36.24%	37.38%	—	—	63.19%	—	—	56.02%
OrientDB	96.63%	96.65%	—	96.55%	96.57%	—	97.96%	97.96%	97.99%	98.03%	98.03%	98.05%
Pentaho Kettle	99.07%	99.09%	99.09%	99.08%	99.10%	99.10%	99.77%	99.78%	99.78%	99.75%	99.77%	99.77%
Pentaho Platform	95.13%	95.22%	95.28%	94.72%	94.82%	94.87%	91.14%	—	89.79%	90.72%	—	89.29%
Pinpoint	96.45%	96.46%	96.45%	96.97%	96.97%	96.96%	98.10%	98.09%	98.16%	98.02%	97.99%	98.09%
PMD	94.07%	92.93%	94.14%	94.94%	—	94.99%	95.39%	95.47%	—	95.76%	95.80%	—
Realm Java	96.31%	94.97%	96.29%	96.28%	94.97%	96.26%	86.40%	86.79%	86.64%	86.61%	86.92%	86.84%
RxJava	86.14%	86.18%	86.90%	83.37%	83.46%	84.62%	87.26%	86.88%	86.12%	84.83%	84.46%	82.98%
Spring Boot	78.69%	80.28%	80.03%	79.01%	80.68%	80.40%	95.67%	96.24%	—	80.27%	96.20%	—
Spring Framework	96.44%	98.25%	98.24%	96.38%	98.19%	98.18%	99.27%	99.26%	99.27%	99.26%	99.26%	99.27%
Spring Security	92.14%	92.18%	92.17%	91.91%	91.94%	91.93%	98.86%	98.86%	98.87%	98.84%	98.84%	98.85%
Tomcat	98.80%	98.79%	98.19%	98.85%	98.84%	98.25%	93.53%	93.38%	93.37%	93.43%	93.27%	93.26%
Tutorials	95.72%	95.80%	96.07%	95.47%	95.59%	95.90%	99.30%	99.38%	99.37%	98.93%	99.04%	99.07%

Note: (i) White: indicates models that do not have a good fit; (ii) Green: indicates models with $\overline{R}^2$ more than 80%; (iii) Yellow: indicates the type(s) of model that better modeled the most of analyzed systems for that metric; (iv) Red: indicates the models selected at Stage 3, which corresponds to the model that better characterizes the evolution of the corresponding metric in the software systems.

TABLE 7 $\overline{R}^2$ values computed from the NOA and NOM models.

	NOA						NOM					
System	lin.	quad.	cub.	log. 1	log. 2	log. 3	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	91.38%	92.34%	—	91.54%	92.66%	—	81.63%	83.25%	—	82.00%	83.53%	—
Antlr4	97.89%	97.89%	97.93%	98.46%	98.46%	98.49%	96.72%	96.96%	97.00%	96.70%	97.21%	97.00%
Arduino	97.73%	97.68%	97.89%	98.45%	98.35%	98.37%	95.14%	95.38%	94.71%	95.81%	96.02%	95.53%
Bazel	95.25%	95.22%	93.80%	95.12%	95.09%	93.60%	97.70%	96.85%	97.72%	97.66%	97.67%	97.68%
Bisq	92.99%	93.62%	—	92.40%	93.16%	—	87.22%	90.38%	85.41%	87.14%	90.23%	85.20%
Buck	90.85%	90.78%	90.71%	91.12%	91.05%	90.99%	98.08%	96.90%	98.10%	97.86%	96.65%	97.87%
CAS	98.77%	97.98%	98.81%	98.96%	98.97%	98.98%	92.93%	93.78%	94.00%	93.10%	94.01%	94.22%
CoreNLP	96.01%	96.09%	96.16%	96.09%	96.15%	96.23%	96.42%	96.63%	96.66%	96.16%	96.42%	96.48%
Dbeaver	98.05%	96.63%	98.13%	97.94%	96.43%	98.02%	98.06%	98.12%	98.17%	98.02%	98.08%	98.13%
Dropwizard	97.68%	97.91%	97.91%	97.89%	98.10%	98.11%	93.95%	94.29%	94.53%	94.41%	94.72%	94.94%
Druid	98.87%	98.97%	99.01%	99.00%	99.03%	99.07%	99.57%	99.57%	99.58%	99.65%	99.65%	99.66%
Eclipse JDT Core	96.95%	96.95%	96.94%	97.32%	97.31%	97.31%	98.62%	98.61%	98.61%	98.91%	98.91%	98.90%
Eclipse PDE UI	87.82%	87.88%	87.04%	86.67%	86.57%	86.13%	87.07%	88.70%	86.37%	86.30%	87.97%	85.41%
Elasticsearch	77.75%	77.69%	80.74%	73.52%	73.42%	76.92%	49.14%	49.61%	40.28%	45.53%	46.00%	36.80%
Equinox Framework	91.05%	91.15%	90.51%	90.53%	90.63%	89.92%	98.42%	98.42%	98.45%	98.52%	98.26%	98.55%
FrameworkBenchmarks	92.25%	89.58%	—	92.32%	89.65%	—	94.92%	94.90%	—	94.96%	95.02%	—
Gocd	44.17%	42.18%	—	51.22%	52.86%	55.73%	51.61%	52.81%	56.45%	52.63%	51.80%	—
Graylog	97.48%	97.61%	97.71%	97.12%	97.65%	97.78%	94.95%	94.98%	95.10%	94.45%	94.49%	94.61%
Guava	99.33%	99.40%	99.75%	99.25%	99.30%	99.25%	99.52%	99.54%	99.56%	99.51%	99.53%	99.54%
Hibernate Orm	89.68%	89.85%	89.94%	90.15%	90.28%	90.40%	97.97%	97.97%	98.05%	97.74%	97.73%	97.82%
J2ObjC	86.88%	87.12%	87.26%	87.57%	87.78%	87.92%	80.30%	80.60%	81.65%	81.96%	82.17%	83.19%
Jabref	99.45%	99.47%	99.31%	99.46%	99.48%	99.48%	98.26%	98.24%	98.22%	98.29%	98.31%	98.27%
Jenkins	95.19%	95.08%	95.17%	95.30%	95.20%	95.29%	96.80%	96.82%	96.88%	96.81%	96.84%	96.89%
Jitsi	91.67%	92.13%	94.47%	91.79%	92.21%	94.53%	92.96%	—	94.74%	92.98%	—	94.75%
JMeter	78.50%	77.54%	78.75%	78.88%	77.84%	78.99%	87.47%	88.25%	—	87.32%	88.14%	—
JUnit 5	95.26%	95.28%	93.75%	95.57%	95.58%	94.21%	98.40%	98.41%	98.47%	98.22%	98.23%	98.29%
K-9 Mail	95.07%	93.75%	95.19%	94.99%	94.62%	95.64%	—	80.18%	—	—	81.85%	—
Kafka	92.64%	—	92.41%	92.88%	—	92.67%	89.74%	—	93.14%	89.85%	—	93.26%
LanguageTool	92.94%	93.04%	92.90%	93.01%	93.08%	92.94%	94.63%	94.65%	95.35%	94.63%	94.65%	95.17%
Lucene	4.58%	9.79%	19.65%	3.31%	6.99%	10.67%	—	—	—	21.42%	—	—
MinecraftForge	60.13%	60.01%	—	37.22%	34.42%	32.66%	43.16%	46.28%	—	32.68%	35.07%	34.81%
Neo4j	98.00%	98.00%	97.99%	97.98%	97.97%	97.96%	84.92%	84.87%	84.92%	85.01%	84.96%	85.01%
Netty	76.23%	77.04%	77.80%	74.82%	75.62%	76.36%	80.18%	80.08%	80.72%	80.24%	80.14%	80.72%
OpenRefine	39.52%	—	40.16%	32.82%	—	32.01%	—	—	34.98%	—	—	—
OrientDB	99.14%	99.14%	99.15%	99.14%	99.14%	99.15%	97.21%	97.22%	97.21%	97.35%	97.36%	97.36%
Pentaho Kettle	97.58%	97.73%	97.47%	97.52%	97.67%	97.40%	98.57%	98.59%	98.59%	98.62%	98.63%	98.63%
Pentaho Platform	86.81%	—	85.35%	86.52%	—	84.97%	94.12%	94.10%	94.11%	93.96%	93.94%	93.94%
Pinpoint	98.44%	98.81%	98.79%	98.44%	98.90%	98.85%	91.87%	91.85%	91.86%	91.46%	91.43%	91.45%

(Continues)

TABLE 7 (Continued)

	NOA						NOM					
System	**lin.**	**quad.**	**cub.**	**log. 1**	**log. 2**	**log. 3**	**lin.**	**quad.**	**cub.**	**log. 1**	**log. 2**	**log. 3**
PMD	87.57%	88.55%	89.52%	87.46%	88.50%	89.46%	96.39%	96.37%	96.43%	97.30%	97.29%	97.34%
Realm Java	96.17%	96.16%	96.16%	96.07%	96.06%	96.07%	95.26%	95.31%	95.10%	95.53%	95.59%	95.53%
RxJava	82.62%	82.96%	82.64%	80.24%	81.02%	80.90%	74.31%	76.54%	74.83%	65.30%	68.57%	66.58%
Spring Boot	80.38%	82.64%	83.32%	80.41%	82.82%	83.50%	97.47%	88.67%	97.55%	97.49%	87.63%	97.58%
Spring Framework	97.07%	97.12%	97.11%	96.98%	97.03%	97.02%	96.37%	98.02%	98.04%	96.30%	97.95%	97.97%
Spring Security	99.43%	99.43%	99.45%	99.42%	99.42%	99.43%	99.53%	99.53%	99.53%	99.52%	99.52%	99.53%
Tomcat	99.86%	99.86%	99.87%	99.87%	99.87%	99.87%	99.68%	99.70%	99.68%	99.68%	99.69%	99.69%
Tutorials	98.47%	98.56%	98.67%	98.29%	98.38%	98.58%	98.02%	98.07%	98.37%	97.69%	97.74%	98.06%

Note: (i) White: indicates models that do not have a good fit; (ii) Green: indicates models with $\overline{R}^2$ more than 80%; (iii) Yellow: indicates the type(s) of model that better modeled the most of analyzed systems for that metric; (iv) Red: indicates the models selected at Stage 3, which corresponds to the model that better characterizes the evolution of the corresponding metric in the software systems.

depth. However, we can not generalize the observations about DIT, LCOM, TCC, NOA, and NOM as a general pattern since our statistical test did not present a significant p-value. The linear model is the one that better explains the evolution of the analyzed software metrics, except for NOC and TCC. A logarithmic at a degree 1 model better models the evolution of NOC. A quadratic model better models the evolution of TCC.

5.2 | Relation between the metrics' evolution

This section answers RQ2.

RQ2. *How does the relation between dimension metrics behave throughout the evolution of software systems?*

We analyze RQ2 for coupling, inheritance hierarchy, and size. For coupling, we quantitatively analyzed the ratio between fan-in and fan-out using the idea of necessary and unnecessary coupling. We followed the same rationale to analyze size, that is, we examined how the proportion of fields and methods of the classes evolve. However, the same reasoning does not apply to the case of inheritance metrics because there is no relevant meaning in the ratio DIT/NOC or NOC/DIT. Hence, for inheritance hierarchy, we carried out a case study to detail how the hierarchy inheritance evolves in terms of depth and breadth.

To organize the discussion of RQ2, we categorize the answers as follows. Section 5.2.1 discusses the relation between fan-in and fan-out. Section 5.2.2 presents the case study of DIT and NOC evolution. Section 5.2.3 describes the evolution of NOA and NOM proportion in the systems over time.

5.2.1 | Evolution of fan-in and fan-out relation

Berard[83] categorizes coupling at the package level into two types: necessary and unnecessary. *Necessary coupling* consists of high fan-in and low fan-out, and *unnecessary coupling* consists of high fan-out and low fan-in. Moreover, according to Lee et al.,[5] a high fan-in may represent a good object design and high reuse since classes at the same package are reused together. In contrast, a high fan-out is undesirable in software because it indicates complexity and low reusability.[83–86]

In this work, we use the necessary and unnecessary coupling to refer to the relation between fan-in and fan-out of a class. In this context, the *necessary coupling* is the ratio of fan-in by fan-out. A high necessary coupling of a class indicates that the class's primary role is a service provider. On the other hand, an *unnecessary coupling* is the ratio of fan-out by fan-in. A high unnecessary class coupling indicates that the central role of the class is a service user. We analyzed which type of coupling stands out during the system's evolution. For this purpose, we computed the necessary and the unnecessary coupling ratios for each system version. We also created charts showing the evolution of these ratios and analyzed if the ratios increased or decreased over time. Table 8 summarizes these two types of coupling behavior.

TABLE 8 Evolution of necessary and unnecessary coupling.

System	NCP direction	UNCP direction
Alluxio	+	–
Antlr4	+	–
Arduino	+	–
Bazel	–	+
Bisq	+	–
Buck	–	+
CAS	–	+
CoreNLP	–	+
Dbeaver	+	–
Dropwizard	–	+
Druid	+	–
Eclipse JDT Core	+	–
Eclipse PDE UI	+	–
Elasticsearch	–	+
Equinox Framework	–	+
Framework Benchmarks	+	–
Gocd	+	–
Graylog	–	+
Guava	+	–
Hibernate Orm	–	+
J2ObjC	+	–
Jabref	–	+
Jenkins	–	+
Jitsi	+	–
JMeter	–	+
JUnit 5	–	+
K-9 Mail	–	+
Kafka	+	–
LanguageTool	–	+
Lucene	+	–
MinecraftForge	–	+
Neo4j	–	+
Netty	–	+
OpenRefine	–	+
OrientDB	–	+
Pentaho Kettle	–	+
Pentaho Platform	–	+
Pinpoint	+	–
PMD	–	+
Realm Java	+	–
RxJava	+	–
Spring Boot	+	–
Spring Framework	–	+
Spring Security	–	+
Tomcat	–	+
Tutorials	+	–

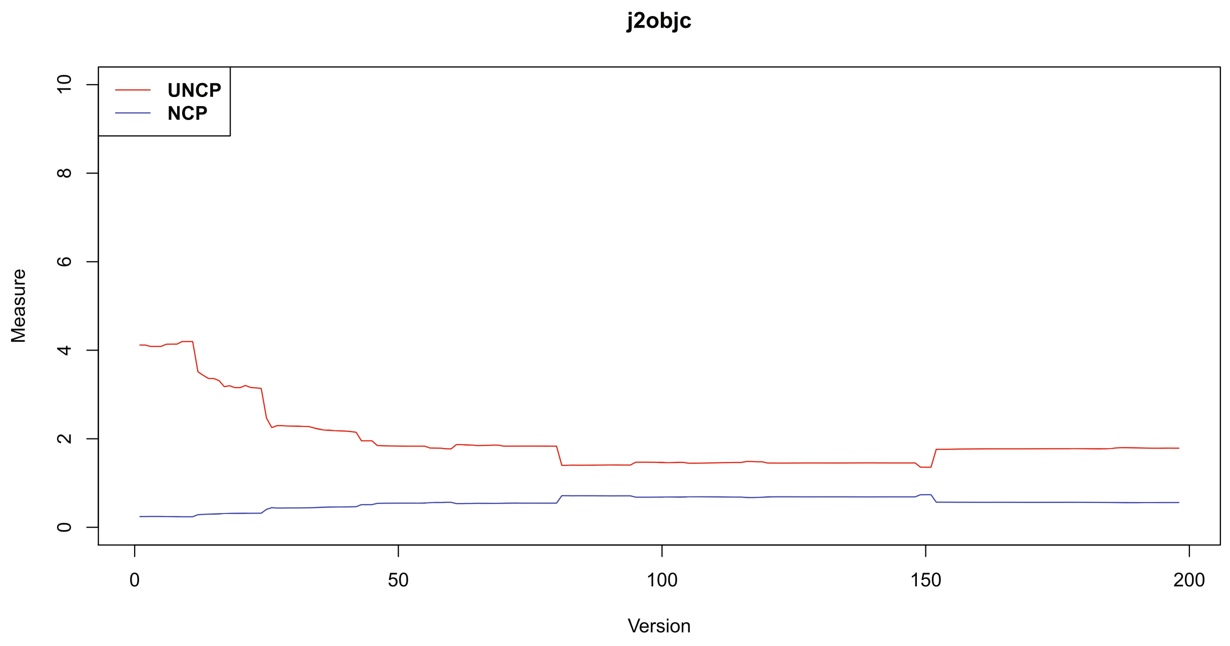


FIGURE 4 Evolution of unnecessary and necessary coupling in J2ObjC.

The acronyms "UNCP" and "NCP" refer to "unnecessary coupling" and "necessary coupling". For the cases where the ratios have increased, we included a "+" signal. When the ratios have decreased, we put a "−" signal. We made the charts available as supplementary material.[‡]

Results in Table 8 show that the necessary coupling has increased in 43% of the systems, while the unnecessary coupling has increased by 57%. This result indicates that classes behave as service users instead of service providers in most analyzed systems.

We visually analyzed the charts of necessary and unnecessary coupling evolution. We observed that unnecessary coupling is usually higher than necessary in all systems. This finding shows that using services from other classes is the principal role of the classes within a system. This analysis shows how prevalent using services is and how it evolves. For instance, Figure 4 exhibits the necessary and unnecessary coupling evolution chart in J2ObjC. In this case, the unnecessary coupling starts high and decreases over the first versions of the system until the 81st version. After that, it remains stable until the 149th version, which presents a very slight decrease. Three versions later, it grows again and remains stable. The necessary coupling in J2ObjC has a smooth increase until the 81st version and remains stable for a period. Between the 149th and 152th versions, the unnecessary coupling suffers a slight variation, but it stabilizes again after the 152th version. Analyzing the evolution of necessary and unnecessary coupling in J2ObjC, we observe that it was developed with a high rate of service user classes. However, service provider classes are introduced over their life until a balance between them is obtained.

As a general conclusion of the analyzed systems, we observed that in 76% of them, necessary and unnecessary couplings do not suffer relevant changes over time. This fact happens with Alluxio, Arduino, Bazel, Bisq, Buck, CAS, CoreNLP,Dbeaver, Druid, Eclipse JDT Core, Eclipse PDE UI, Elasticsearch, Equinox Framework, Gocd, Hibernate Orm, Jabref, Jenkin, Jitsi, JMeter, JUnit 5, K-9 Mail, LanguageTool, Lucene, Neo4j, Netty, OpenRefine, Pentaho Kettle, Pentaho Platform, PMD, Realm Java, RxJava, Spring Boot, Spring Framework, Spring Security, Tomcat. In Antlr4, FrameworkBenchmarks, Guava, J2ObjC, Pinpoint, Tutorials, the unnecessary coupling decreased over the first releases until stabilization in the last releases. This fact means that, in the beginning, the new classes inserted in the system were more service users than providers.

In Dropwizard and Kafka, the unnecessary coupling had similar behavior but in different directions. In the beginning, the unnecessary coupling varied a little, and in some releases after, it suffered a relevant change. This change was a drop in Dropwizard, while in Kafka, it was a climb. After that, the unnecessary coupling is still stable over the

[‡]https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch.

subsequent releases. When analyzing `Graylog` and `MinecraftForge`, we noticed a high variation of unnecessary and necessary couplings at the beginning of these systems' lifetime. The fact that may explain these variations is the occurrence of several changes, which may have led to an instability of the two types of couplings at the beginning of these two systems' lifetime. However, this variation stabilizes around the 100th version and keeps going this way over time. Finally, `OrientDB` behaved differently from other systems. The unnecessary coupling proliferates in its first versions. However, after the 100th version, it decreased and returned to the unnecessary coupling level that `OrientDB` had at the beginning of its evolution.

Summary of RQ2–Coupling. The unnecessary coupling is higher than the necessary coupling since the first release of a system, meaning that the rate of classes behaving as service users is higher than the service providers. In most cases, the system's evolution does not change the relation between fan-in and fan-out.

5.2.2 | Evolution of DIT and NOC relation

This section presents a case study with `Eclipse JDT Core` to analyze how the hierarchy inheritance grows over the software evolution. We decided to use `Eclipse JDT Core` in this case study for two reasons: (i) it composes the dataset used in this work, and (ii) it is a well-known and representative Java software system in practice. To perform this analysis, we compared the DIT and NOC time series for each class of this software system and observed how they have evolved over the versions.

We observed that the inheritance hierarchy has grown in `Eclipse JDT Core` from the bottom, that is, classes have been added in the leaves of the tree with minimal impact on DIT. Figure 5 shows an example of this behavior.

Figure 5 shows the evolution of DIT and NOC metrics in `org.eclipse.jdt.internal.compiler.ast.Expression` `Eclipse JDT Core`. This result indicates that this class remained in the same level of inheritance tree during the system's lifetime. On the other hand, sub-classes were often added to this class over its evolution. This effect shows that the inheritance hierarchy has consistently grown from the bottom in this software system.

Summary of RQ2–Inheritance Hierarchy. Comparing the evolution of DIT and NOC in `Eclipse JDT Core`, we conclude that the inheritance hierarchy in this system has grown from the bottom, that is, classes have been added to the tree's leaves. Such a finding characterizes how this software system's inheritance hierarchy has evolved. However, this preliminary analysis needs a deeper investigation to generalize this pattern for all object-oriented Java software systems.

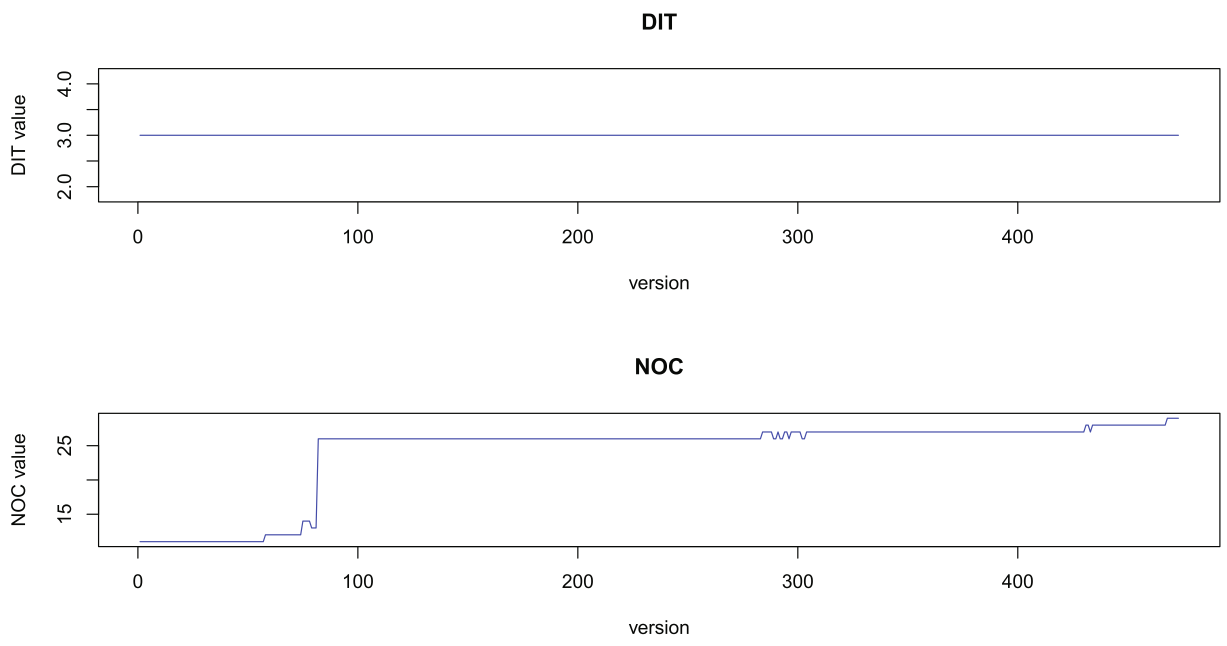


FIGURE 5 Detailing of inheritance hierarchy evolution for a class in `Eclipse JDT Core`.

5.2.3 | Evolution of NOA and NOM relation

This part analyzes how the proportion between the number of attributes (NOA) and the number of methods (NOM) occurs and evolves over the software evolution. For this purpose, we divided NOA by NOM and computed the global proportion of these metrics. The global proportion consists of using the global values of NOA and NOM, extracted by taking the arithmetic average and dividing the global NOA by the global NOM, obtaining the global proportion of these metrics. Table 9 summarizes the behavior of the proportions. For the cases where the proportions have increased, we put a "+" signal. In the cases where the ratios have decreased, we included a "−" signal. The charts that show the evolution of the NOA and NOM proportions are available as supplementary material.[§]

Analyzing Table 9, we notice that 35% of the proportions have increased while 65% have decreased. Such observation shows that the number of methods has increased more than the number of attributes over the systems' lifetime. Observing the proportions chart, we identify that in 80% of the systems, the global NOA and NOM proportions vary between 20% and 60%. Such observation suggests an oscillation range where the NOA and NOM proportion remains over the systems' lifetime. The systems where this behavior happens are `Alluxio`, `Bazel`, `Bisq`, `Buck`, `CoreNLP`, `Dbeaver`, `Dropwizard`, `Druid`, `J2ObjC`, `Eclipse JDT Core`, `Eclipse PDE UI`, `Elasticsearch`, `Equinox Framework`, `Gocd`, `Guava`, `Hibernate Orm`, `Jabref`, `Jenkins`, `Jitsi`, `Jmeter`, `JUnit 5`, `K-9 Mail`, `LanguageTool`, `Lucene`, `Neo4j`, `Netty`, `OpenRefine`, `OrientDB`, `Pentaho Platform`, `Pinpoint`, `PMD`, `Realm Java`, `RxJava`, `Spring Boot`, `Spring Framework`, `Spring Security`, `Tomcat`, `Tutorials`.

In the case of the `Bazel`, `CoreNLP`, `Dbeaver`, `Druid`, `Gocd`, `Guava`, `Hibernate Orm`, `Jenkins`, `Jitsi`, `Jmeter`, `LanguageTool`, `Lucene`, `OpenRefine`, `Pentaho Platform`, `Pinpoint`, `Spring Framework`, `Tomcat` systems, we observe that they had a small variation in their NOA and NOM proportion. Besides, they remain almost stable over their lifetime. `Framework Benchmarks` and `Pentaho Kettle` are systems where the proportion exceeded 60%. `Framework Benchmarks` starts with 80% of the proportion. However, it decreases over its lifetime and stabilizes at 60%. `Pentaho Kettle` starts with 50% of proportion, which increases until ≈70%, and decreases again, stabilizing at 60%.

`Arduino`, `Antlr4`, `CAS`, `Graylog`, `Kafta`, and `MinecraftForge` are the ones that had a significant change in the NOA and NOM proportion, which successive refactoring or additions of new functionalities may have caused. Figure 6 shows an example of this proportion in `Arduino`. Analyzing this figure, we can observe that the number of attributes was higher at the beginning of the system than the number of methods. This behavior suggests evidence of the presence of Data Class in its internal components. *Data class* is a bad smell that characterizes a class with a high quantity of data and none or low features that process the data.[87] Classes with this symptom may indicate behavior incorrectly since they display much internal implementation and are often manipulated in too much detail by other classes. In the 47th version, the NOA and NOM proportion decreases and maintains this behavior until it stabilizes close to the 250th version. We can realize that many points of the chart in Figure 6 had a sudden drop in the proportion value. This action may have probably been generated by refactoring or restructuring the system's internal architecture to improve its internal quality and allow the inclusion of new features.

Summary of RQ2–Size. The NOA and NOM proportion tends to decrease over time, and such a finding shows that NOM tends to increase at a higher rate than NOA, and the systems tend to grow more in features than data. Besides, the NOA and NOM proportion varies from 20% to 60% and remains inside this interval over the software lifetime.

5.3 | Analysis of growth and decrease of metrics' values

This section answers RQ3.

RQ3. *Which proportion of classes within the software system affects the growth and the decrease of the dimensions?*

We analyzed the percentages of classes from the systems directly responsible for increasing and decreasing the metrics' values of the analyzed dimensions. We carried out trend analysis in the time series of the systems' classes and computed the percentage of classes whose metrics have increased and decreased over time. Besides, we calculated the rates of types responsible for increasing and decreasing these metrics values and summarized them in Figures 7–10. We removed the ghost classes and classes under ten observations from our trend analysis, and we subtracted both of them from the system's whole classes to compute the percentages.

[§]https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch.

TABLE 9 Evolution of NOA and NOM proportion.

System	Proportion direction
Alluxio	+
Antlr4	–
Arduino	–
Bazel	–
Bisq	–
Buck	–
CAS	+
CoreNLP	–
Dbeaver	+
Dropwizard	–
Druid	+
Eclipse JDT Core	+
Eclipse PDE UI	+
Elasticsearch	–
Equinox Framework	–
FrameworkBenchmarks	–
Gocd	–
Graylog	–
Guava	–
Hibernate Orm	+
J2ObjC	+
Jabref	–
Jenkins	–
Jitsi	+
JMeter	–
JUnit 5	–
K-9 Mail	–
Kafka	–
LanguageTool	–
Lucene	–
MinecraftForge	–
Neo4j	+
Netty	–
OpenRefine	–
OrientDB	–
Pentaho Kettle	–
Pentaho Platform	+
Pinpoint	+
PMD	–
Realm Java	+
RxJava	–
Spring Boot	–
Spring Framework	+
Spring Security	+
Tomcat	–
Tutorials	+

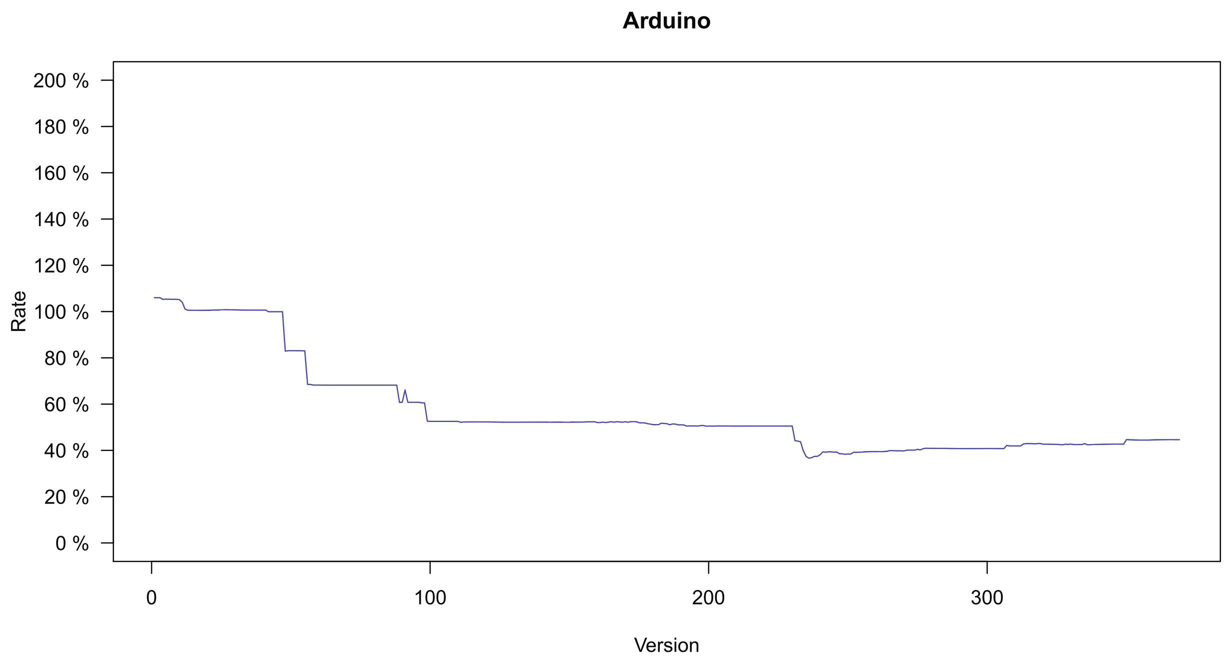


FIGURE 6 NOA and NOM proportion evolution in Arduino.

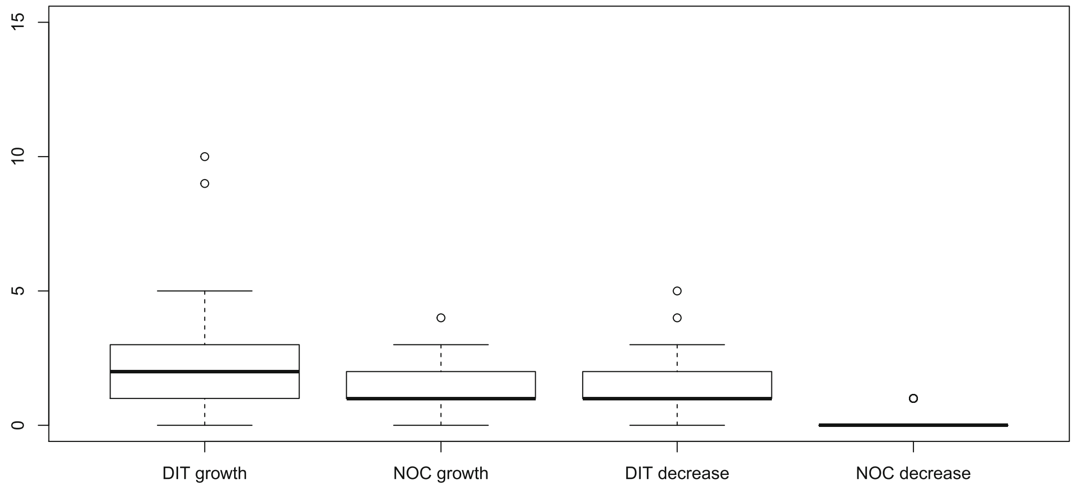


FIGURE 7 Distribution of classes that affect DIT and NOC growth and decrease.

Dimensions growth. Figures 7–10 show the percentage of classes with an increasing tendency in their metrics. In this analysis, we considered only the valid classes, that is, those that are not ghosts and have more than ten observations. Fan-out, fan-in, NOM, and NOA presented the highest percentages: 37%, 24%, 23%, and 15%, respectively. DIT and NOC showed a meager rate, no more than 10%, and 4%, that is, very few classes have these metrics increased over the system evolution. This fact is not surprising since just a few percentages of classes are usually involved in the inheritance tree. Besides, some software maintenance activities, for example, refactoring, have contributed to not increasing the inheritance metrics' values over the software evolution. Regarding cohesion, no more than 2% of the classes have such metrics increased.

Dimensions Decreasing. Figures 7–10 show that a small group of classes decreased their metrics over the software evolution. We found that no more than 14%, 13%, and 10% of the classes contribute to decreased Fan-out, NOM, and NOA metrics. When analyzing fan-in, DIT, TCC, NOC, and LCOM, we found even smaller percentages in this sequence: 6%, 5%, 3%, 1%, and 1%.

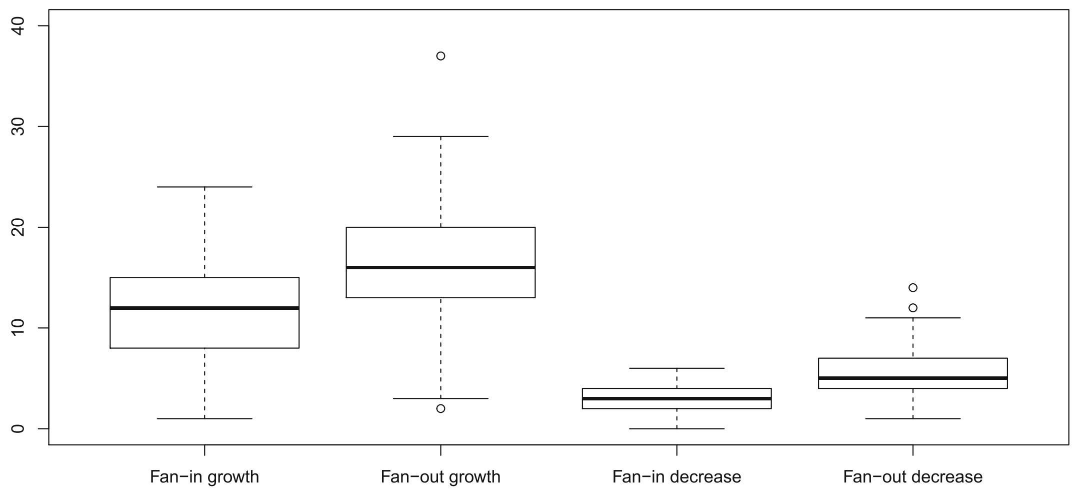


FIGURE 8 Distribution of classes that affects FAN-IN and FAN-OUT growth and decrease.

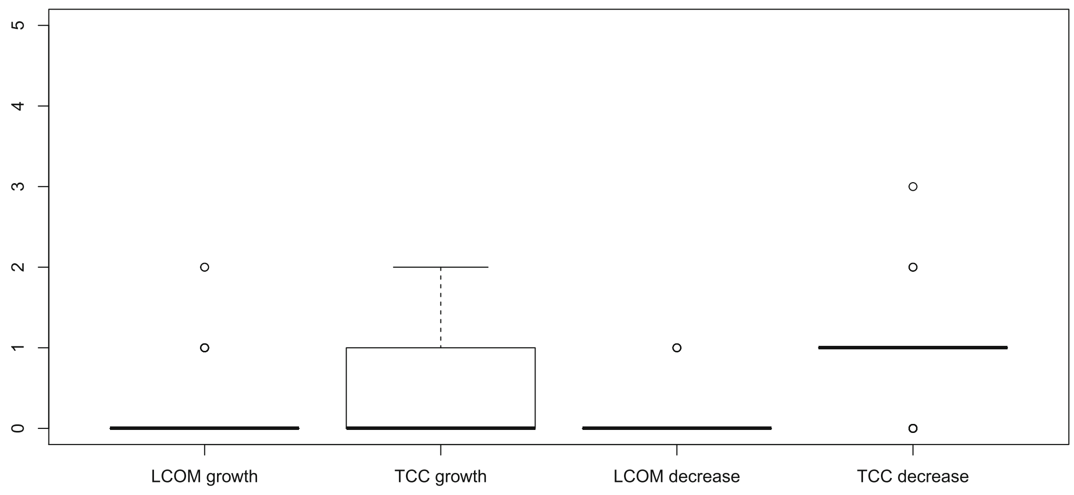


FIGURE 9 Distribution of classes that affects LCOM and TCC growth and decrease.

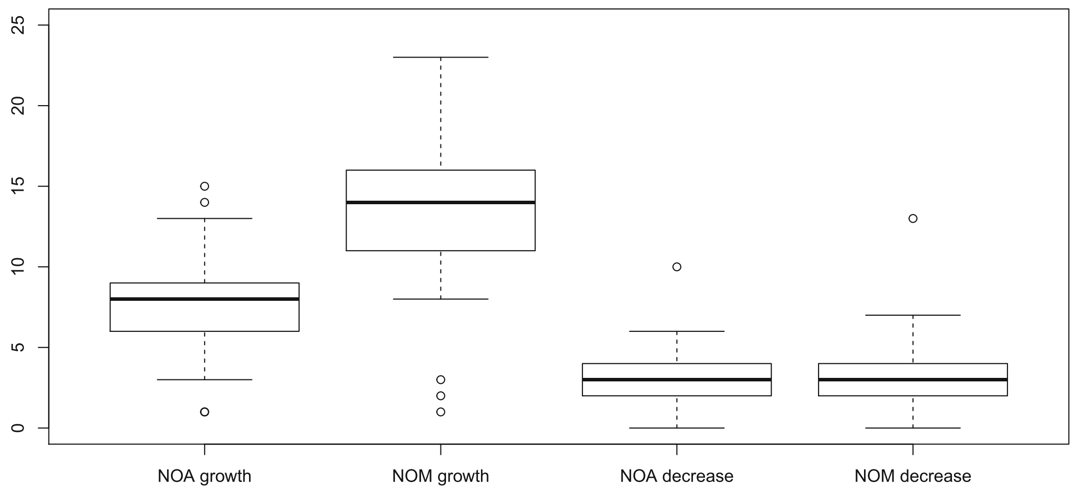


FIGURE 10 Distribution of classes that affects NOA and NOM growth and decrease.

Growth versus Decrease. We analyzed the intersection trend results for the metrics of the same dimension to identify if they evolve following a well-defined pattern. We considered the following behaviors in our analysis:

1. first and second metrics grow,
2. first and second metrics decrease,
3. the first metric grows while the second metric decreases, and
4. the first metric decreases, and the second metric grows.

Table 10 summarizes the results obtained for these cases. Each group of four columns contains a pair of metrics separated by a dash, for example, "Fan-in–Fan-out". The metric before the dash corresponds to the first metric, and the one after the dash corresponds to the second metric.

This table shows that all intersection cases are rare for inheritance hierarchy and cohesion dimensions, and the percentages obtained for inheritance hierarchy are at most 1%. There is no case of an intersection trend pattern between LCOM and TCC. The cases *(ii)*, and *(iv)* in the coupling, and *(ii)*, *(iii)* and *(iv)* in class size dimensions are rare to occur since they also presented 3% as a maximum percentage. Case *(i)* in both coupling and class size dimensions and case *(iii)* in coupling are the behaviors with more chance to occur since they presented percentages of 12%, 10%, and 8%, respectively. Although case *(i)* in both coupling and class size and case *(iii)* in coupling presented the highest percentages, the results suggest that the classes that directly affect the evolution dimensions metrics do not follow a combined growth and decrease pattern.

Summary of RQ3. The increase of fan-out, fan-in, NOM, and NOA is defined by 37%, 24%, 23%, and 15%, respectively. A small percentage of classes have increased LCOM, TCC, DIT, and NOC. In terms of decreasing, only a tiny percentage of classes have a metric decreased over the software evolution. The evolution of the metrics regarding each internal dimension does not follow a related pattern.

6 | FORECASTING ANALYSIS

This section answers **RQ4.** *How well can our time series-based approach predict software evolution?*

This research question aims to evaluate the prediction accuracy of the best models we found in Section 5.1 by following our evaluation protocol. As stated in Step 2 of Section 4, in this analysis, we used the data of the test subset to evaluate the accuracy of the predictions generated by the obtained models. The objective is to assess if the best models fitted to the training subset are also good at producing forecasts for the test subset. In addition, we built prediction intervals to evaluate the reliability of the obtained predictions.

To evaluate the accuracy of the models, we considered two types of forecast: (i) short-term forecast and (ii) long-term forecast. Short-term forecast refers to the capacity of the model to predict values for a short period, using one-step ahead forecasts. Long-term forecast refers to the ability of the fitted model to obtain reliable predictions for distant points in time.

Then, we carried out the following process. Initially, we used the fitted model to obtain the forecasts for the test subset. We compared the predicted values with the real values to evaluate how close the predicted value was to the correct value. We must highlight that using the whole training set, we used the structure and coefficients obtained in the modeling process to perform the predictions for the long-term forecast. However, when performing the short-term prediction, we kept the original design of the models. We updated the coefficients, incorporating the real value of the data set, one at a time, to obtain a new prediction.

To evaluate the precision of the forecasts obtained by the fitted models, we used the predicted mean squared error (PMSE). We decided to use PMSE because it is one of the most common metrics used in the literature to analyze the accuracy of forecasts.[76] This metric compares the predicted value to the real value, and the difference, called predicted error (e_i), is extracted. The predicted mean squared error (PMSE) is calculated according to the following formula:

$$\text{PMSE} = \frac{\sum_{i=1}^{n} e_i^2}{n} \tag{1}$$

In the formula, n is the number of calculated predictions. We extracted a PMSE measure for each time series of our dataset using the best model extracted by our approach. We also plotted a prediction chart for the analyzed time series, comparing the real and the predicted values. We did not include all charts in this paper because we analyzed a

TABLE 10 Intersection percentages of the trend results.

	DIT – NOC				Fan-in – Fan-out				LCOM – TCC				NOA – NOM			
System	i	ii	iii	iv	i	ii	iii	iv	i	ii	iii	iv	i	ii	iii	iv
Alluxio	0%	0%	0%	0%	3%	0%	1%	0%	0%	0%	0%	0%	4%	0%	0%	1%
Antlr4	0%	0%	0%	0%	8%	1%	4%	1%	0%	0%	0%	0%	6%	1%	0%	1%
Arduino	0%	0%	0%	0%	4%	1%	2%	1%	0%	0%	0%	0%	5%	2%	0%	2%
Bazel	0%	0%	0%	0%	6%	1%	1%	1%	0%	0%	0%	0%	6%	1%	1%	1%
Bisq	0%	0%	0%	0%	4%	1%	1%	1%	0%	0%	0%	0%	3%	1%	0%	1%
Buck	0%	0%	0%	0%	6%	0%	1%	1%	0%	0%	0%	0%	6%	1%	0%	1%
CAS	0%	0%	0%	0%	11%	1%	3%	3%	0%	0%	0%	0%	5%	4%	3%	2%
CoreNLP	0%	0%	0%	0%	6%	0%	1%	0%	0%	0%	0%	0%	5%	0%	0%	0%
Dbeaver	0%	0%	0%	0%	8%	0%	1%	1%	0%	0%	0%	0%	7%	1%	0%	0%
Dropwizard	0%	0%	0%	0%	7%	0%	1%	0%	0%	0%	0%	0%	6%	1%	0%	1%
Druid	0%	0%	0%	0%	6%	0%	0%	0%	0%	0%	0%	0%	4%	1%	0%	2%
Eclipse JDT Core	1%	0%	0%	0%	10%	0%	1%	1%	0%	0%	0%	0%	8%	1%	0%	1%
Eclipse PDE UI	0%	0%	0%	0%	2%	0%	1%	0%	0%	0%	0%	0%	3%	1%	0%	0%
Elasticsearch	0%	0%	0%	0%	6%	1%	2%	1%	0%	0%	0%	0%	6%	1%	1%	1%
Equinox Framework	0%	0%	0%	0%	12%	1%	1%	1%	0%	0%	0%	0%	9%	1%	1%	1%
FrameworkBenchmarks	0%	0%	0%	0%	3%	1%	1%	1%	0%	0%	0%	0%	3%	1%	1%	1%
Gocd	0%	0%	0%	0%	4%	1%	1%	1%	0%	0%	0%	0%	4%	1%	1%	1%
Graylog	0%	0%	0%	0%	6%	0%	1%	1%	0%	0%	0%	0%	5%	1%	0%	1%
Guava	0%	0%	0%	0%	5%	0%	1%	1%	0%	0%	0%	0%	2%	1%	0%	1%
Hibernate Orm	0%	0%	0%	0%	10%	1%	1%	1%	0%	0%	0%	0%	7%	1%	1%	1%
J2ObjC	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	3%	1%	0%	1%
Jabref	0%	0%	0%	0%	5%	1%	1%	1%	0%	0%	0%	0%	5%	2%	1%	1%
Jenkins	1%	0%	0%	0%	9%	1%	1%	1%	0%	0%	0%	0%	9%	1%	0%	1%
Jitsi	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%
JMeter	0%	0%	0%	0%	3%	0%	0%	0%	0%	0%	0%	0%	5%	0%	0%	1%
JUnit 5	0%	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	4%	1%	0%	1%
K-9 Mail	0%	0%	0%	0%	2%	1%	1%	1%	0%	0%	0%	0%	5%	1%	1%	0%
Kafka	0%	0%	0%	0%	9%	0%	1%	0%	0%	0%	0%	0%	10%	1%	0%	1%
LanguageTool	0%	0%	0%	0%	7%	0%	0%	0%	0%	0%	0%	0%	6%	0%	0%	1%
Lucene	0%	0%	0%	0%	4%	1%	1%	1%	0%	0%	0%	0%	5%	2%	1%	1%
MinecraftForge	0%	0%	0%	0%	3%	1%	0%	1%	0%	0%	0%	0%	5%	1%	1%	0%
Neo4j	0%	0%	0%	0%	5%	1%	1%	1%	0%	0%	0%	0%	5%	1%	1%	1%
Netty	1%	0%	0%	0%	7%	1%	1%	1%	0%	0%	0%	0%	5%	1%	0%	1%
OpenRefine	0%	0%	0%	0%	4%	2%	8%	0%	0%	0%	0%	0%	4%	1%	0%	1%
OrientDB	0%	0%	0%	0%	7%	0%	1%	0%	0%	0%	0%	0%	4%	0%	0%	0%
Pentaho Kettle	0%	0%	0%	0%	6%	0%	1%	0%	0%	0%	0%	0%	5%	0%	1%	0%
Pentaho Platform	0%	0%	0%	0%	5%	1%	1%	0%	0%	0%	0%	0%	6%	1%	0%	0%
Pinpoint	0%	0%	0%	0%	3%	1%	2%	1%	0%	0%	0%	0%	4%	2%	0%	1%
PMD	0%	0%	0%	0%	7%	2%	1%	1%	0%	0%	0%	0%	5%	1%	0%	1%
Realm Java	0%	0%	0%	0%	4%	0%	0%	1%	0%	0%	0%	0%	4%	0%	1%	0%
RxJava	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Spring Boot	0%	0%	0%	0%	4%	1%	1%	1%	0%	0%	0%	0%	4%	2%	0%	1%
Spring Framework	0%	0%	0%	0%	8%	0%	0%	1%	0%	0%	0%	0%	5%	1%	0%	1%
Spring Security	0%	0%	0%	0%	4%	1%	1%	1%	0%	0%	0%	0%	5%	1%	1%	1%
Tomcat	0%	0%	0%	0%	7%	1%	1%	1%	0%	0%	0%	0%	7%	3%	1%	1%
Tutorials	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

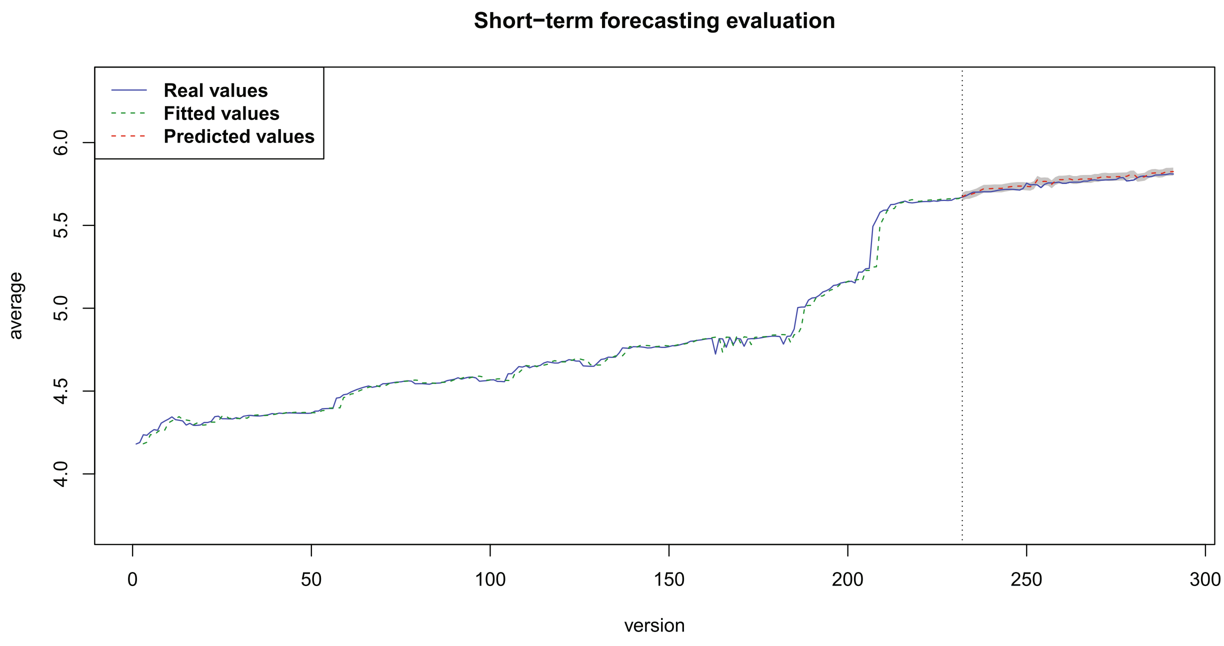


FIGURE 11 Evaluation of the short-term prediction for the model extracted for `spring-framework` regarding the Fan-out metric.

large amount of data and extracted many prediction models. However, we made them available online as supplementary material.[¶]

To discuss the efficiency of our approach and answer this research question, we divided the discussion of this section as follows. First, we show and detail an example of a model extracted for a time series of a particular metric. After, we discuss the efficiency of our approach by comparing the errors presented by the best prediction model we identified with the models regarding the other types we have evaluated for both short-term and long-term forecasts.

Example of a prediction model. In this part, we report the forecasts obtained from a model extracted by our approach to exemplify its efficiency. The analysis is performed by presenting a prediction chart that compares the model fit, the forecasts, and the prediction intervals. The case we discussed here is the model extracted for the fan-out metric in `spring-framework`.

Figures 11 and 12 show the prediction charts we generated after evaluating the data from test phase for both short-term and long-term prediction of the model reported here. In both charts, the blue line represents the real value, that is, the original values for each version in the time series. The green line represents the fitted values, and the red line refers to the predicted values. The vertical dotted line divides the data we used for extracting the model and the data we used for measuring the model's accuracy concerning predictions. The shaded grey area represents the 95% prediction interval.

The analysis of the charts shows that the extracted model is very accurate, as it produced reasonable estimates for both short-term and long-term predictions. For the short-term prediction, Figure 11, the forecasts were very close to the real values.

For the long-term prediction, Figure 12, the forecasts obtained for the first observations were better, as expected. We can observe that the predicted value was very close to the real value in the predictions performed from the 233th to 260th observations. However, although the forecasts started to lose precision after the 260th observation, this behavior is expected because the further the observation is from the training data, the more the estimation quality is reduced. However, although the predicted values stay distant from the real values, they are inside our prediction interval.

Comparative analysis of the obtained models. In this part of our analysis, we compared the PMSE generated from all models described in Section 5.1. Figure 13 summarizes the distribution of the errors for the short-term forecasts, and Figure 14 shows the distribution of the PMSE values obtained for the long-term forecasts.

Analyzing Figures 13 and 14, we observe that, in general, our approach obtained accurate prediction models with low prediction error. As the linear model was the one that better fitted to the time series pattern of most of the analyzed

[¶]https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch.

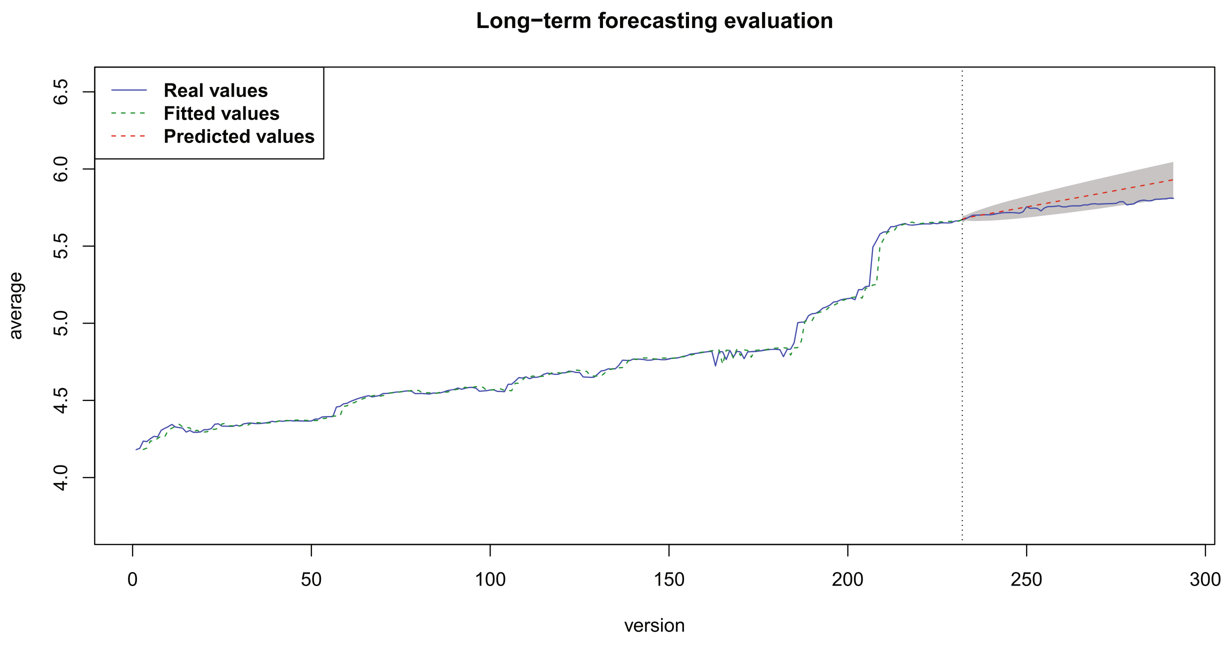


FIGURE 12 Evaluation of the long-term prediction for the model extracted for `spring-framework` regarding the Fan-out metric.

metrics in Section 4, we can observe that it has PMSE close to 0 and it is the one that contains the fewest outliers both for long-term and short-term forecasts. Figure 13 shows the results for the short-term forecasts. In these results, PMSE usually reaches values between 0 and 1. As aforementioned, we expected that evaluating the short-term forecasts would lead to low values. As the coefficients of the models are updated for each step ahead prediction, abrupt changes that might occur in the time series pattern are quickly identified. Therefore, it is easier for the model to detect the change and to produce accurate forecasts.

Observing the assessment of the long-term forecast in Figure 14, we can see an increase in the PMSE compared with the short-term forecast. However, when analyzing the distribution of the best metrics models, that is, linear for DIT, fan-in, fan-out, LCOM, NOA, and NOM, logarithmic at Degree 1 for NOC, and quadratic for TCC, we observe that the distribution of their PMSE values is very close to 0, except in some cases, especially for the cubic and logarithmic at degree 3. This fact shows that most of the time, our models generate forecasts close to the real values of the test subset, even for horizons far from the last period of the training subset.

Summary of RQ4. Our time series-based approach can produce accurately predicted values for short or long-term forecasts.

7 | SOFTWARE EVOLUTION PROPERTIES

This section compiles our results in ten properties of software evolution related to coupling, inheritance hierarchy, cohesion, and size.

First–Coupling grows linearly over time. The arithmetic average of fan-in and fan-out usually increases over time. Moreover, the linear model is the one that best describes their evolution in software systems. This property contains a relevant practical application. As linear models are a simple and easy-to-implement methodology, representing and characterizing these dimensions is straightforward. Thus, researchers who wish to model the evolution of coupling in other systems can take advantage of the results obtained in this work and build linear models to explain the behavior of fan-in and fan-out.

Second–Unnecessary coupling is continuously higher 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. The unnecessary coupling is higher than the necessary coupling since the first release of a software system. This fact means that most classes in a system are service users. The consequence of this property is that, as a software system evolves, we should pay more attention to classes that provide services, as the impact of changes on them tends to increase over time. Researchers

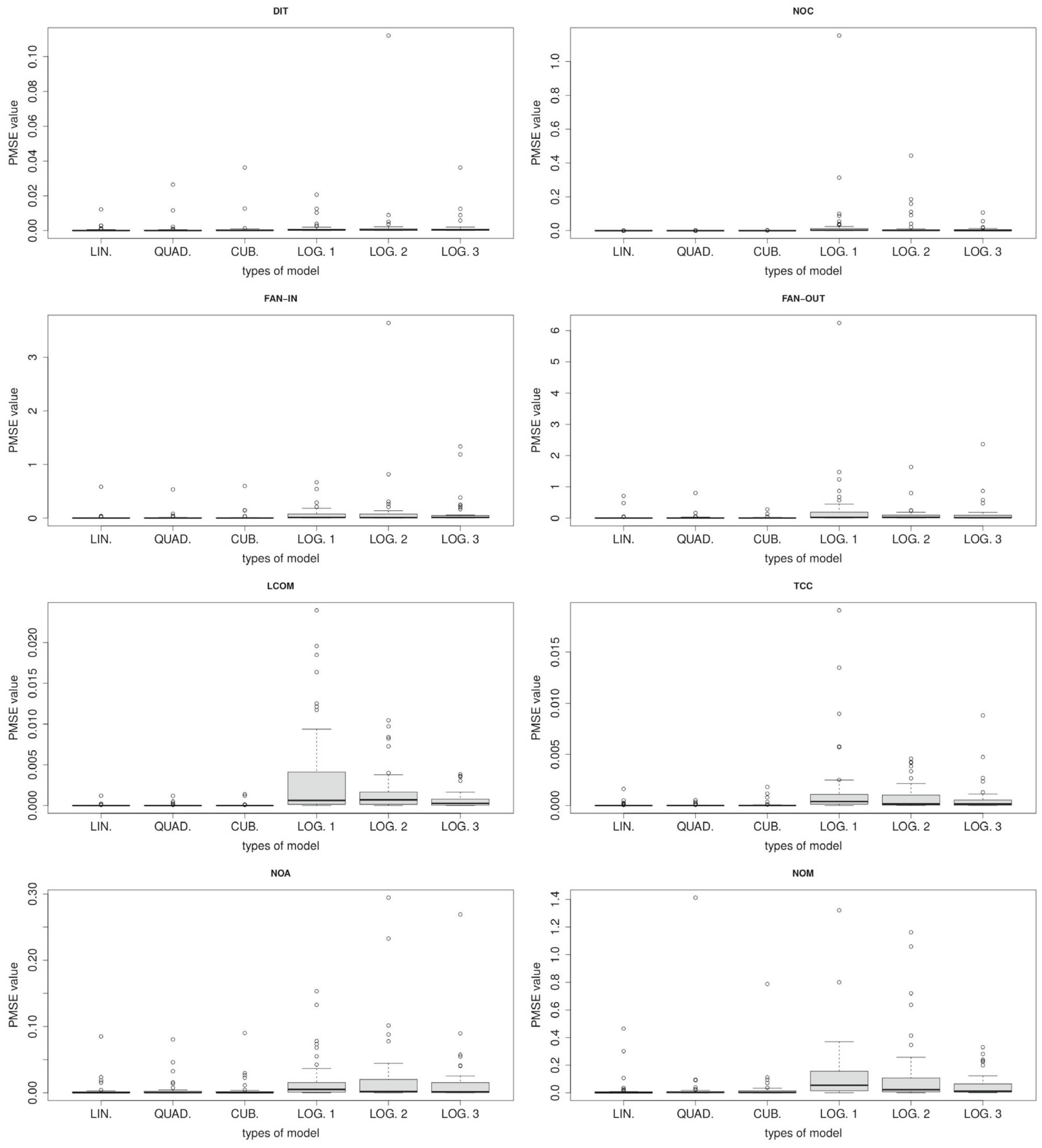


FIGURE 13 Distribution of PMSE of the prediction models extracted for the short-term forecast.

can apply this property in practice to improve the maintainability and quality of software systems. As we have identified profiles of classes that have increased and become the software more complex, researchers can avoid them during the development process, or they can be the focus of attention during the process of refactoring or software maintenance.

Third–Complexity is introduced in the first versions of a system. We observed that unnecessary coupling is higher than necessary coupling since the first system versions. Based on this analysis and the quality indication pointed out by the literature, we concluded that the system's initial version is already complex. This finding contradicts the assumption that complexity is inserted in software systems over their evolution. However, this finding is consistent with the study of Tufano et al.,[88] whose main conclusion is that bad smells have been introduced in software systems since

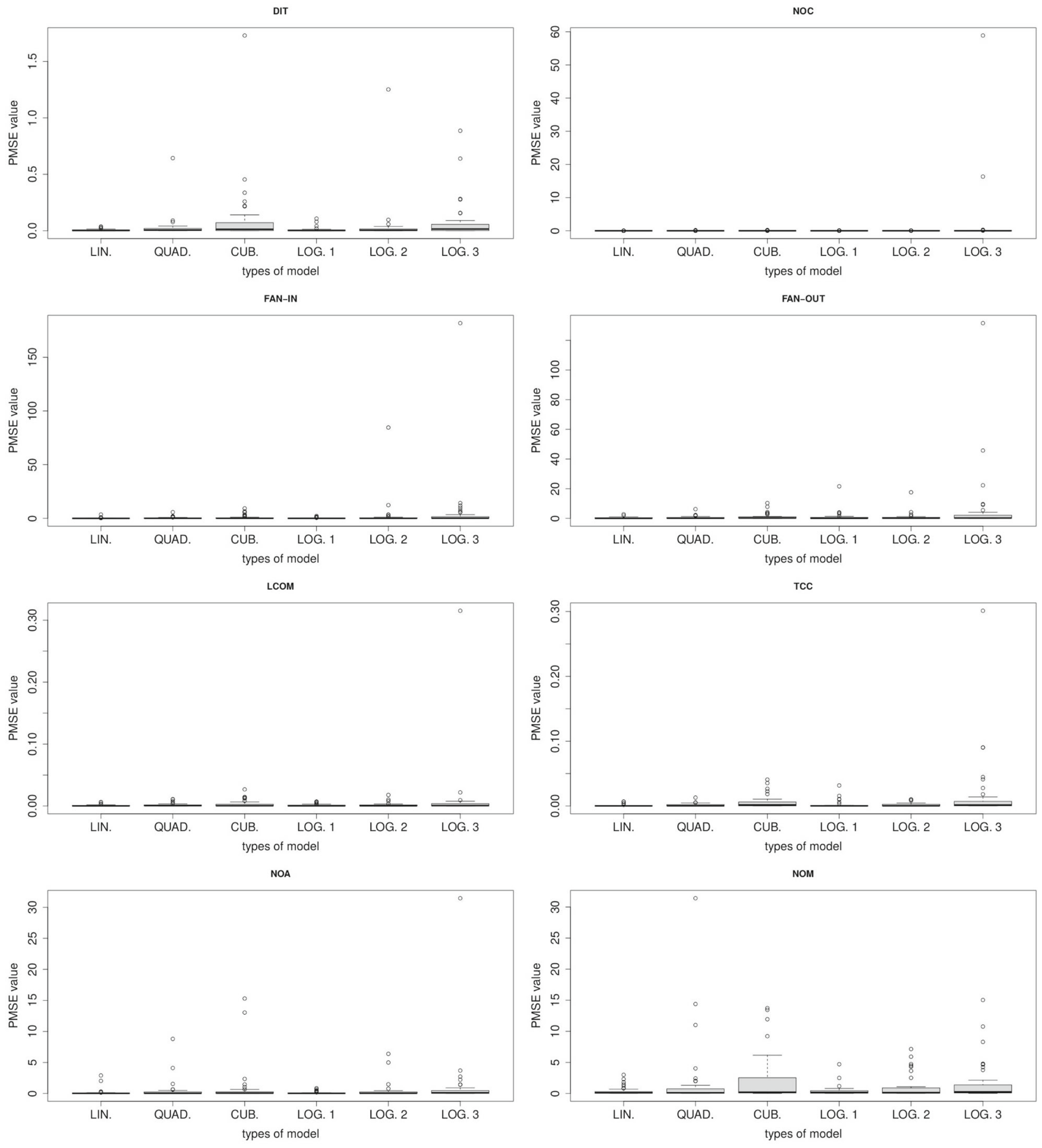


FIGURE 14 Distribution of PMSE of the prediction models extracted for the long-term forecast.

their first versions. The main benefit of this property in practice is determining the period where complex classes are usually inserted inside the software system. Considering this knowledge, developers can pay more attention to the initial phase of development and plan actions to avoid introducing complex classes in this phase. As another practical application, this property indicates the need to develop refactoring tools and techniques to be applied from the beginning of the systems' life cycle.

Fourth–Inheritance hierarchy tends to increase linearly in-depth and logarithmic in-breadth. The depth of the inheritance hierarchy is given by DIT, and the breadth by NOC. Although our results did not indicate a well-defined pattern for the global inheritance hierarchy in terms of growth and decrease, we observed a tendency for the global inheritance hierarchy to grow in breadth and depth over time. We performed a case study with `Eclipse JDT Core` to analyze how this increasing pattern occurs in the system and found that the inheritance tree has grown from the bottom of the system, that is, due to the inclusion of new sub-classes at the bottom of the inheritance tree. These findings lead to two main conclusions: (i) as the DIT's growth is linear and NOC is logarithmic, the depth of the tree tends to increase more than its breadth, that is, developers apply more specialization of classes at the bottom of the tree than of classes in the middle of the tree; (ii) the linear and the logarithmic model indicates that DIT and NOC grow slightly and, hence, the number of classes included in the inheritance tree does not tend to be high.

Fifth–Cohesion tends to increase slightly. LCOM and TCC are metrics that characterize cohesion in software systems. However, they measure cohesion differently. We analyzed both metrics in this study. When LCOM decreases, it indicates that the software is becoming more cohesive. On the other hand, when TCC increases, it indicates the software is becoming more cohesive. We found a tendency for LCOM to decrease and TCC to increase over time in most analyzed systems. However, LCOM tends to evolve linearly, whereas TCC tends to evolve quadratically. Such a difference may be due to the forms in which the metrics are calculated: LCOM considers all the methods of the classes, whereas TCC considers only the public methods. Despite the difference between the evolution model for LCOM and TCC, their behavior suggests that the systems become more cohesive over time, in contrast with the intuitive notion that as a system evolves, its classes lose cohesion. A possible explanation for this behavior may be the refactoring performed in the systems over their life cycle.

Sixth–The systems have increased coupling but not necessarily reduced cohesion. Coupling has an increasing pattern in the analyzed systems. Intuitively, it may be expected that the systems would become more complex and their cohesion would decay. However, our analysis shows that the internal cohesion has improved over time instead of worsening in most analyzed systems. A possible explanation for this behavior may be the refactoring in open-source software. In this case, the refactoring practices have been effective in controlling the classes' cohesion but less effective in controlling the coupling among them.

Seventh–Class size evolves linearly in terms of data and features. The evolution of size metrics tends to grow linearly regarding data and features. We did not find a well-defined evolution pattern for the analyzed size metrics. However, we identified evidence of a slight tendency for the global class size to decrease in terms of data and features. The practical application of this property, as already pointed out for coupling, is that linear models are straightforward to implement.

Eighth–The proportion of the number of attributes concerning the number of methods in a class decreases over time. We identified that the global NOA and NOM ratio decreases over time, and the NOA and NOM proportion varies from 20% to 60% most of the time. This property suggests that the contract of the systems' internal components increases over time because more methods have been designed to process the internal attributes of the classes and provide features to the user classes.

Ninth–A significant percentage of classes directly contributes to the coupling and size evolution. In contrast, a small portion directly impacts on the inheritance hierarchy and cohesion evolution. Applying the trend analysis, we found that about 30% and 23% of classes contribute directly to the growth of coupling and size dimensions, respectively. However, the group contributing to these dimensions' decrease is not greater than 10%. Regarding inheritance hierarchy and cohesion, we found that no more than 10% and 2% contribute to their growth, while no more than 5% and 4% contribute to their decrease. Therefore, the practical application that this property brings is that the evolution of software systems has been controlled by a particular group of systems that represents approximately a quarter of the software system are probably are classes containing business rules of the software system.

Tenth–There is no association between the software metrics evolution for the internal dimensions. We analyzed the percentages of four association cases between metrics of the same dimension in growth and decrease. We found low values for all examined associations. The association we obtained with a better percentage was fan-in and fan-out growing together. However, the maximum ratio obtained from this case was 12%, which did not represent a quarter of the systems. Then, such findings suggest that the pair of metrics analyzed for each dimension are unrelated, and the metrics do not follow a combined pattern over the evolution of the software systems.

8 | PRACTICAL IMPLICATIONS

The dataset produced and applied in this work may be useful in software engineering studies. As we found in the literature review, there needs to be more public, large, and updated datasets for supporting software evolution analysis. We constructed a novel dataset of 46 Java software systems for our studies. Although we analyzed six metrics in this study, we constructed a comprehensive dataset with 46 software metrics. The dataset contains data on metrics' time series until 2020 and is publicly available. Hence, other researchers can use it for conducting case studies or empirical studies focused on software evolution.

We also proposed a method for modeling the evolution of software systems. Researchers and practitioners can use our method to model particular software systems. Besides, as our approach is generic, practitioners can also use it to understand how particular characteristics of their software systems are evolving and, hence, define actions to preserve their quality and maintainability. Besides, our method gives a landscape vision of how a system evolves regarding an internal quality attribute. This vision may allow the practitioners to identify points in which the analyzed system has become complex, difficult to maintain, or even less cohesive and, hence, apply refactoring or other actions to improve their quality.

Besides the steps of our model documented in this paper, we built a semi-automatic approach that allows practitioners and researchers to apply our method to analyze how the structure of a software system is evolving. In this approach, the user collects the necessary data for running the method. The only requirement of our semi-automatic approach is that the software system to be analyzed is kept in a repository on GitHub. We made these scripts publicly available on GitHub.[#] With the scripts we used to build the dataset, the user can extract the time series from the software system following the instructions registered on the link repository. After preparing the data, the user can run the time series method proposed in this paper by using other scripts we implemented and made publicly available on GitHub.[||] They were implemented in R programming language and ran each step of our method automatically.

We carried out an empirical study applying the software evolution method and aiming to characterize the evolution of Java open-source systems. The first practical implication of this study is that the proposed method efficiently supports developers and researchers in studying the evolution of software systems and extracting relevant insights about them. Another practical implication of this empirical study refers to the software evolution properties. Such properties bring a fine-grained view regarding the evolution of internal aspects of Java open-source systems. They can also help developers to plan strategies for controlling some aspects, for example, coupling, and avoiding that they will become serious problems of complexity and harm the maintainability of the systems. Another implication of our empirical study for practitioners is the application of those properties for easing maintenance activities. For instance, one of our properties indicates that a small group of classes in the software system is responsible for the increase or decrease of an internal attribute. Having this fact in mind, practitioners can use the trend analysis phase of our method, which is already automated in R by our scripts, to identify the components that directly impact the increase and decrease of that internal attribute and focus their actions of refactoring or redesign only in those necessary components.

Finally, the last practical implication of this work is the support for building prediction models provided by our software evolution method. By the analysis we carried out in Section 6, we showed that our method is efficient for building models able to carry out accurate predictions. In this way, practitioners can use our method to extract prediction models of particular software systems. This model is helps anticipate essential decisions about the software system. For instance, by using these models, a designer can identify that according to the evolution pattern of a software system, its coupling will increase a lot in some weeks ahead, making it very complex. Knowing this fact, the developers can also plan and apply solutions in the internal structure of the software system so that its complexity does not increase as much as projected by the prediction model.

9 | THREATS TO VALIDITY

We defined a trend analysis with statistical trend tests to identify the classes that impact the growth and decrease of software metrics. Using these tests may be considered a threat to validity since statistical tests may be susceptible to misapplied errors. We defined three relevant and useful tests existing in the literature to mitigate this threat. We also established criteria for a trend, which must be indicated by at least two of the three tests.

We studied the evolution of four relevant internal dimensions in 26 open-source Java systems. Although we considered many systems, our results reflect the evolution of just open-source Java systems. We may not generalize our results for systems to other domains and contexts, for example, proprietary systems or systems written in other object-oriented

[#] https://github.com/BrunoLSousa/DSTool.

[||] https://github.com/BrunoLSousa/TSAnalysisMethod.

languages. However, since Java is the most used language in open-source software development, the results presented in this study bring relevant insight into the evolution of open-source systems.

We used linear regression techniques to model the time series. Although regression techniques are often used in this kind of data,[40–42,89–93] the presence of interventions or autocorrelation in the time series may turn the results spurious if not included in the model. Time series may also contain missing observations, that is, observations without a measure, making it challenging to fit a model to the data. We applied a reconstruction data approach to ensure we would not have time series without observations. We also carried out intervention and residual analyses after using linear regression to treat autocorrelation and ensure that the models reasonably adjusted to the time series to mitigate this threat.

10 | RELATED WORK

This section discusses related work and groups them according to the software metrics they consider.

10.1 | Coupling

Eski and Buzluca[33] studied the relationship between two object-oriented coupling metrics and software changes. The analyzed metrics were coupling between objects (CBO) and responses for a class (RFC). They concluded that a metric might differ in its change-proneness according to the software domain. However, the RFC presented the most significant correlation with changes in different projects, and it may be helpful to determine critical parts of the systems. Abuasad and Alsmadi[34] analyzed the correlation of coupling metrics with faults. They pointed out a high correlation between these two factors indicating that coupling metrics are handy in characterizing the software quality and maintenance effort. Couto et al.[35] also investigated the relation between software metrics and the occurrence of bugs and concluded that coupling metrics are reliable indicators of a defect. Singh and Ahmed[36] analyzed the relation between coupling metrics and changes and identified a positive correlation between these aspects. They also found that high coupling might be less fault-prone and may not trigger more changes than classes with low coupling. Such findings contradict well-known studies, such as References 94–97. Our work differs from these works because they investigated coupling metrics' relation with changes and failures. We study and detail the evolution of this aspect in object-oriented software systems.

10.2 | Cohesion

Eski and Buzluca[33] analyzed the relationship between two object-oriented cohesion metrics and software changes. They studied metrics concerning the lack of cohesion in methods (LCOM) and cohesion among methods (CAM), and their results did not indicate any relationship between cohesion metrics and change-proneness. Alenezi and Zarour[98] investigated the relationship between modularity and cohesion property and confirmed this relationship as positive in their results. The authors concluded that cohesion metrics could usefully measure modularity in software systems. Alenezi and Zarour[98] also analyzed the evolution of cohesion in two open-source systems and found that this property has not improved in these systems, impairing their modularity over time. Our work differs from the previous studies on cohesion because we analyze its evolution and detail its pattern.

10.3 | Inheritance hierarchy

Meyer[37] investigated the use of inheritance in object-oriented software. They proposed a catalog with 12 different forms to use this mechanism, discussing the forms and the scenarios where they are applied. Daly et al.;[30] Cartwright;[31] and Harrison et al.[32] investigated the impact of depth of inheritance on maintenance and extracted different conclusions. Daly et al.[30] indicated that inheritance hierarchy harms maintenance. Cartwright[31] suggested that inheritance hierarchy positively impacts this aspect. Harrison et al.[32] studied the impact of inheritance hierarchy on modifiability and understandability in C++ systems. It concluded that systems without inheritance are more straightforward to modify and understand than those containing inheritance levels. Tempero et al.[27] analyzed the use of inheritance in Java developers' programs. They found that approximately 75% of the analyzed systems are defined using inheritance, and the classes' inheritance tree tends not to be high in-depth and breadth. Nasseri et al.[26] investigated the evolution of inheritance hierarchy in Java systems and realized that it grows breadth-wise instead of depth-wise. They also identified that classes are more prone to be added at levels 1 and 2 than at other deeper levels. Our work differs from these works because we describe how the inheritance hierarchy evolves instead of analyzing its impact on external quality attributes.

10.4 | Size

One of the most common literature attempts is to describe how the size evolves. Many works have tried to define a pattern for size evolution. However, there is no agreement between them. While some works have indicated that the size in open-source systems increases in a super-linear way,[16,21,25,28] others have found that the size property grows in a sub-linear pattern.[19,22,23] There are still studies that have concluded that size increases linearly[20] or following a Pareto distribution.[24] Capiluppi and collaborators[17,18] also realized that the size property follows a similar pattern for three different perspectives: the size of files, number of files, and lines of code. Hatton et al.[29] analyzed the size evolution to quantify the source code's growth rate in open-source software. They concluded that the systems tend to double the average rate of produced source code every 42 months. Our work differs from the previous works because we analyze the evolution of size from the data and feature perspectives and not from the lines of code perspective.

10.5 | Prediction

The software evolution literature has often used the time series approach to extract and propose prediction models. For instance, some studies in the literature have focused on proposing prediction models aiming to predict defects.[38–43] They mainly differ in the approach used to build the model. Couto et. al.,[38] used Granger causality test. Raja et al.,[39] Yazdi et al.,[73] and Wu et al.[43] applied ARIMA. Graves et al.[40] and Arisholm and Briand[41] used linear regression techniques. Although they used different strategies, they generally obtained accurate predictions with low error values. Besides, Antoniol et al.[45] proposed a method based on time series and ARIMA for monitoring and predicting clones' evolution over a software system lifetime. They concluded that their model was accurate and produced an average error of less than 4%. Caprio et al.[46] also built a model using time series and ARIMA, aiming to estimate software systems' size and complexity. They identified an accuracy with low error values and some peaks. Amin et al.[47] extracted an ARIMA model from time series aiming to forecast quality of service (QoS) attributes. They evaluated their model using QoS datasets from the real world and identified that it outperforms other popular existing ARIMA models and improves the forecasting accuracy by an average 35.4%[44] applied the ARIMA time series technique for modeling and forecasting change requests per unit of size of large open-source projects. The results obtained from the model assessment confirmed that it could effectively predict and identify trends with a low error rate. Although the related works have applied techniques for analyzing and modeling time series, our work differs from them because we have analyzed characteristics not yet explored by them. Besides, while the related works have proposed prediction models following a methodology focused on a particular characteristic, we have proposed a general method that can be applied to any other type of internal characteristic of the software since time series models are used in the analysis.

11 | CONCLUSION

This work presents an empirical study on coupling, inheritance hierarchy, cohesion, and class size evolution in object-oriented systems. In this work, we (i) investigated how these internal dimensions evolve and describe the pattern of their evolution; (ii) analyzed how the relationship between dimension metrics behave over the system's evolution; (iii) identified the portion of classes existing in the systems that directly affect the evolution of these dimensions; and (iv) analyzed the accuracy of time series approach we proposed in the context of prediction of software evolution.

We defined a two-phase method based on time series analysis to analyze software evolution, where the first phase extracts the global time series and models them by applying linear regression to identify the type of model that better describes the evolution of the dimension metrics. The second one uses trend tests to detect the classes that affect the evolution of the dimensions metrics.

Based on the results of our analysis, we identified ten properties that describe and characterize the software evolution, bringing new insights into how the internal structure of object-oriented systems evolves. Some of these properties are:

1. the dimensions have evolved linearly, except the inheritance hierarchy regarding the depth and cohesion in terms of TCC.
2. among the analyzed dimensions, coupling, and breadth of inheritance hierarchy has grown over time
3. inheritance hierarchy has presented evidence of decreasing in depth.
4. Although we could not generalize the behavior of cohesion evolution because the statistical tests did not show a significant difference, a descriptive analysis has shown improved systems cohesion over time
5. size has presented evidence of increasing in terms of data and features
6. most of the system's classes do not change their metrics values, and consequently, the increase or decrease of the dimensions is affected by a small group of classes; and
7. evidence of non-association between analyzed dimension metrics.

We applied our approach to predict software evolution. The results indicate that our method is efficient, and the models extracted can generate good short-term and long-term forecasts.

Knowing how the software evolution properties evolve in current software development environments brings insights into how the internal structure of software systems evolves and provides a background for software engineering decisions. The approach we defined and applied in this work may be used to analyze other software dimensions and increase knowledge of how software systems evolve.

Further studies are essential to (i) investigate the evolution of other internal characteristics not considered in our work; (ii) replicate these analyses in other contexts; (iii) conduct qualitative analyses to identify factors that affect the evolution of the dimensions we considered in this work; (iv) analyze the impact of these dimensions' evolution in software maintenance; and (v) construct a tool by joining all the scripts we implemented in this work aiming to allow the use of our method in an even more practical way.

AUTHOR CONTRIBUTIONS

Bruno L. Sousa contributed to the definition of the methodologies, building of the dataset, implementation and definition of the analysis method, conduction of the experiments, analysis and interpretation of the results, and writing of the article. **Mariza A. S. Bigonha** contributed to the definition of the topic, definition of the methodologies, analysis of the experiments, interpretation of the results, and writing and revision of the article. **Kecia A. M. Ferreira** contributed to the definition of the topic, definition of the methodologies, analysis of the experiments, interpretation of the results, and writing and revision of the article. **Glaura, C. Franco** contributed to the entire statistical part, assisting in the choice of methods, tests and creation of the analysis method proposed in the article. She also contributed to the analysis and interpretation of the results and the writing and revision of the article.

ACKNOWLEDGMENTS

This work was supported by CAPES, CNPq, FAPEMIG, UFMG Programming Language Research Group, UFMG Graduate Program in Computer Science, CEFET-MG, and the Department of Statistics from UFMG.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Supplementary Material SPE Research at https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch.

ORCID

Bruno L. Sousa https://orcid.org/0000-0002-8217-3524

REFERENCES

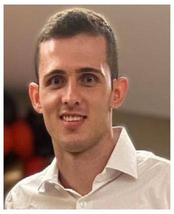
1. Mens T, Guéhéneuc Y, Fernández-Ramil J, D'Hondt M. Guest editors' introduction: software evolution. *IEEE Softw*. 2010;27(4):22-25. doi:10.1109/MS.2010.100
2. Lehman M, Ramil J, Wernick P, Perry D, Turski W. Metrics and laws of software evolution-the nineties view. Proceedings Fourth International Software Metrics Symposium, IEEE. 1997:20-32.
3. Erlikh L. Leveraging legacy system dollars for e-business. *IT Profess*. 2000;2(3):17-23.

4. Sommerville I. *Software Engineering*. 9th ed. Pearson; 2012.
5. Lee Y, Yang J, Chang KH. Metrics and evolution in open source software. Paper presented at: Seventh International Conference on Quality Software (QSIC 2007) IEEE. 2007:191-197.
6. Mens T, Fernández-Ramil J, Degrandsart S. The evolution of eclipse. Paper presented at: IEEE International Conference on Software Maintenance IEEE. 2008:386-395.
7. Xie G, Chen J, Neamtiu I. Towards a better understanding of software evolution: an empirical study on open source software. Paper presented at: 2009 IEEE International Conference on Software Maintenance IEEE. 2009:51-60.
8. Businge J, Serebrenik A, Brand VDM. An empirical study of the evolution of eclipse third-party plug-ins. Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) ACM. 2010:63-72.
9. Israeli A, Feitelson D. The Linux kernel as a case study in software evolution. *J Syst Softw*. 2010;83(3):485-501.
10. Alenezi M, Almustafa K. Empirical analysis of the complexity evolution in open-source software systems. *Int J Hybrid Inf Technol*. 2015;8(2):257-266.
11. Zhang J, Sagar S, Shihab E. The evolution of mobile apps: an exploratory study. Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile ACM. 2013:1-8.
12. Li D, Guo B, Shen Y, Li J, Huang Y. The evolution of open-source mobile applications: an empirical study. *J Softw: Evol Process*. 2017;29(7):e1855.
13. Gezici B, Tarhan A, Chouseinoglou O. Internal and external quality in the evolution of mobile software: an exploratory study in open-source market. *Inf Softw Technol*. 2019;112:178-200.
14. Barry E, Kemerer C, Slaughter S. How software process automation affects software evolution: a longitudinal empirical analysis. *J Softw Maintenance Evol: Res Pract*. 2007;19(1):1-31.
15. Gonzalez-Barahona J, Robles G, Herraiz I, Ortega F. Studying the laws of software evolution in a long-lived FLOSS project. *J Softw: Evol Process*. 2014;26(7):589-612.
16. Godfrey M, Tu Q. Evolution in open source software: a case study. Proceedings 2000 International Conference on Software Maintenance IEEE. 2000:131-142.
17. Capiluppi A, Morisio M, Ramil J. The evolution of source folder structure in actively evolved open source systems. Paper presented at: 10th International Symposium on Software Metrics, 2004. Proceedings. IEEE. 2004:2-13.
18. Capiluppi A, Morisio M, Ramil J. *Structural Evolution of an Open Source System: A Case Study*. Vol 12. IEEE; 2004:172-182.
19. Capiluppi A, Ramil J. *Studying the Evolution of Open Source Systems at Different Levels of Granularity: Two Case Studies*. IEEE; 2004:113-118.
20. Robles G, Amor J, Gonzalez-Barahona J, Herraiz I. Evolution and growth in large libre software projects. Paper presented at: Eighth International Workshop on Principles of Software Evolution (IWPSE'05) IEEE. 2005:165-174.
21. Herraiz I, Robles G, González-Barahona J, Capiluppi A, Ramil J. Comparison between SLOCs and number of files as size metrics for software evolution analysis. Paper presented at: Conference on Software Maintenance and Reengineering (CSMR'06) IEEE. 2006 8.
22. Izurieta C, Bieman J. The evolution of FreeBSD and Linux. Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering ACM. 2006:204-211.
23. Capiluppi A, Fernandez-Ramil J, Higman J, Sharp H, Smith N. An empirical study of the evolution of an agile-developed software system. Proceedings of the 29th International Conference on Software Engineering IEEE Computer Society. 2007:511-518.
24. Herraiz I, Gonzalez-Barahona J, Robles G. Towards a theoretical model for software growth. Paper presented at: Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007) IEEE. 2007:21.
25. Koch S. Software evolution in open source projects–a large-scale investigation. *J Softw Maintenance Evol: Res Pract*. 2007;19(6):361-382.
26. Nasseri E, Counsell S, Shepperd M. An empirical study of evolution of inheritance in Java OSS. Paper presented at: 19th Australian Conference on Software Engineering (Aswec 2008) IEEE. 2008:269-278.
27. Tempero E, Noble J, Melton H. How do Java programs use inheritance? An empirical study of inheritance in Java software. Paper presented at: European Conference on Object-Oriented Programming Springer. 2008:667-691.
28. Gonzalez-Barahona J, Robles G, Michlmayr M, Amor J, German D. Macro-level software evolution: a case study of a large software compilation. *Empir Softw Eng*. 2009;14(3):262-285.
29. Hatton L, Spinellis D, Genuchten M. The long-term growth rate of evolving software: empirical results and implications. *J Softw: Evol Process*. 2017;29(5):e1847.
30. Daly J, Brooks A, Miller J, Roper M, Wood M. Evaluating inheritance depth on the maintainability of object-oriented software. *Empir Softw Eng*. 1996;1(2):109-132.
31. Cartwright M. An empirical view of inheritance. *Inf Softw Technol*. 1998;40(14):795-799.
32. Harrison R, Counsell S, Nithi R. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J Syst Softw*. 2000;52(2-3):173-179.
33. Eski S, Buzluca F. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. Paper presented at: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops IEEE. 2011:566-571.
34. Abuasad A, Alsmadi IM. Evaluating the correlation between software defect and design coupling metrics. Paper presented at: 2012 International Conference on Computer, Information and Telecommunication Systems (CITS) IEEE. 2012:1-5.
35. Couto C, Silva C, Valente MT, Bigonha R, Anquetil N. Uncovering causal relationships between software metrics and bugs. Paper presented at: 2012 16th European Conference on Software Maintenance and Reengineering IEEE. 2012:223-232.

36. Singh G, Ahmed MD. Effect of coupling on change in open source Java systems. Proceedings of the Australasian Computer Science Week Multiconference ACSW'17. ACM. Association for Computing Machinery; New York, NY, USA. 2017.
37. Meyer B. The many faces of inheritance: a taxonomy of taxonomy. *Computer*. 1996;29(5):105-108.
38. Couto CFM. Predicting Software Defects with Causality Tests. PhD thesis. UFMG, Belo Horizonte, Minas Gerais 2013.
39. Raja U, Hale D, Hale J. Modeling software evolution defects: a time series approach. *J Softw Maintenance Evol*. 2009;21(1):49-71.
40. Graves TL, Karr AF, Marron JS, Siy H. Predicting fault incidence using software change history. *IEEE Trans Softw Eng*. 2000;26(7):653-661.
41. Arisholm E, Briand LC. Predicting fault-prone components in a java legacy system. Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering ACM. 2006:8-17.
42. Ratzinger J, Pinzger M, Gall H. EQ-mine: Predicting short-term defects for software evolution, 4422 LNCS. 2007 Braga, Portugal: 12-26.
43. Wu W, Zhang W, Yang Y, Wang Q. Time series analysis for bug number prediction. Paper presented at: The 2nd International Conference on Software Engineering and Data Mining IEEE. 2010:589-596.
44. Kenmei B, Antoniol G, Di Penta M. Trend analysis and issue prediction in large-scale open source systems. Paper presented at: 2008 12th European Conference on Software Maintenance and Reengineering IEEE. 2008:73-82.
45. Antoniol G, Casazza G, Di Penta M, Merlo E. Modeling clones evolution through time series. Proceedings IEEE International Conference on Software Maintenance. ICSM 2001 IEEE. 2001:273-280.
46. Caprio F, Casazza G, Di Penta M, Villano U. Measuring and predicting the Linux kernel evolution. Proceedings of the International Workshop of Empirical Studies on Software Maintenance Citeseer. 2001.
47. Amin A, Grunske L, Colman A. An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering IEEE. 2012:130-139.
48. Lorenz M, Kidd J. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc; 1994.
49. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Trans Softw Eng*. 1994;20(6):476-493.
50. Bieman JM, Kang BK. Cohesion and reuse in an object-oriented system. Proceedings of the 1995 Symposium on Software Reusability SSR'95. ACM. Association for Computing Machinery; New York, NY, USA. 1995:259-262.
51. Bär H, Bauer M, Ciupke O, et al. *The Famoos Object-Oriented Reengineering Handbook*. SCG FAMOOS; 1999.
52. Panas T, Lincke R, Lundberg J, Lowe W. A qualitative evaluation of a software development and Re-engineering project. Paper presented at: 29th Annual IEEE/NASA Software Engineering Workshop IEEE. 2005:66-75.
53. SEDataset. A Time Series-Based Dataset of Open-Source Software Evolution. https://brunolsousa.github.io/software-evolution-dataset/index.html 2021.
54. Darcy D, Daniel S, Stewart K. Exploring complexity in open source software: evolutionary patterns, antecedents, and outcomes. Paper presented at: 2010 43rd Hawaii International Conference on System Sciences IEEE. 2010:1-11.
55. Yacoub SM, Ammar HH, Robinson T. Dynamic metrics for object oriented designs. Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403) IEEE. 1999:50-61.
56. Arisholm E, Briand LC, Foyen A. Dynamic coupling measurement for object-oriented software. *IEEE Trans Softw Eng*. 2004;30(8):491-506.
57. Mitchell A, Power JF. Run-time cohesion metrics for the analysis of Java programs. Tech. Rep. NUIM-CS-TR-2003-08, National University of Ireland; Kildare, Ireland 2003.
58. Mitchell A, Power JF. Run-time cohesion metrics: an empirical investigation. Proc. the International Conference on Software Engineering Research and Practice; Las Vegas, USA. 2004:532-537.
59. Mitchell A, Power JF. Using object-level run-time metrics to study coupling between objects. Proceedings of the 2005 ACM Symposium on Applied Computing. 2005:1456-1462.
60. Mitchell Á, Power JF. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Sci Comput Program*. 2006;59(1-2):4-25.
61. Hassoun Y, Johnson R, Counsell S. A dynamic runtime coupling metric for meta-level architectures. Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. IEEE. 2004:339-346.
62. Hassoun Y, Johnson R, Counsell S. Empirical validation of a dynamic coupling metric. Birkbeck College London, Technical Report BBKCS-04-03 2004.
63. Hassoun Y, Counsell S, Johnson R. Dynamic coupling metric: proof of concept. *IEE Proc-Softw*. 2005;152(6):273-279.
64. Gupta N, Rao P. Program execution based module cohesion measurement. Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001) IEEE. 2001:144-153.
65. Ralph P, Tempero E. Construct validity in software engineering research and software metrics. Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018 EASE'18. ACM. Association for Computing Machinery; New York, NY, USA. 2018:13-23.
66. D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. Paper presented at: Msr 2010 IEEE. 2010:31-41.
67. Vasa R, Lumpe M, Jones A. Helix - Software Evolution Data Set. http://www.ict.swin.edu.au/research/projects/helix 2010.
68. Tempero E, Anslow C, Dietrich J, et al. The Qualitas corpus: a curated collection of Java code for empirical studies. Paper presented at: Software Engineering Conference (APSEC), 2010 17th Asia Pacific (APSEC, ed.) IEEE. 2010:336-345.
69. Couto C, Maffort C, Garcia R, Valente MT. COMETS: a dataset for empirical research on software evolution using source code metrics and time series analysis. *ACM SIGSOFT Softw Eng Notes*. 2013;38(1):1-3.
70. Sousa BL, Bigonha MA, Ferreira KA. Analysis of coupling evolution on open source systems. Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse. 2019:23-32.
71. Aniche M. Java code metrics calculator (CK). 2015 https://github.com/mauricioaniche/ck/

72. Draper N, Smith H. *Applied Regression Analysis*. Wiley; 1981.
73. Yazdi H, Mirbolouki M, Pietsch P, Kehrer T, Kelter U. Analysis and prediction of design model evolution using time series. Paper presented at: International Conference on Advanced Information Systems Engineering Springer. 2014:1-15.
74. Box G, Jenkins G. *Time Series Analysis: Forecasting and Control*. Holden-Day; 1976.
75. Wei W. *Time Series Analysis: Univariate and Multivariate Methods*. Pearson Addison Wesley; 2006.
76. Bowerman B, O'Connell R. *Forecasting and Time Series: an Applied Approach*. Duxbury classic series. Duxbury Press; 1993.
77. Cowpertwait PSP, Metcalfe AV. *Introductory Time Series with R*. 1st ed. Springer Publishing Company, Incorporated; 2009.
78. Miles J. *R Squared, Adjusted R Squared*. American Cancer Society; 2014.
79. Kendall MG. *Rank Correlation Methods*. LDN: Charless Griffin; 1975.
80. Morettin P, Toloi C. *Time Series Analysis*. ABE-Fisher Project Edgard Blucher; 2006 (In portuguese).
81. Hamed K, Rao A. A modified Mann-Kendall trend test for autocorrelated data. *J Hydrol*. 1998;204(1-4):182-196.
82. Conover WJ. *Practical Nonparametric Statistics*. Vol 350. John Wiley & Sons; 1998.
83. Berard EV. *Essays on Object-Oriented Software Engineering*. Vol 1. Prentice-Hall, Inc; 1993.
84. Booch G. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc; 1991.
85. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc; 1996.
86. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR; 2003.
87. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley; 1999.
88. Tufano M, Palomba F, Bavota G, et al. When and why your code starts to smell bad. Paper presented at: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. 1. IEEE. 2015:403-414.
89. Ramil JF, Lehman MM. Metrics of software evolution as effort predictors-a case study. Paper presented at: Icsm. 2000:163-172.
90. Capiluppi A. Models for the Evolution of OS Projects. International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. IEEE. 2003:65-74.
91. Koch S. Evolution of open source software systems–a large-scale investigation. Proceedings of the 1st International Conference on Open Source Systems. 2005:148-153.
92. Shatnawi R, Li W. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J Syst Softw*. 2008;81(11):1868-1882.
93. Kirbas S, Sen A, Caglayan B, Bener A, Mahmutogullari R. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. 2014.
94. Abreu FB, Melo W. Evaluating the impact of object-oriented design on software quality. Proceedings of the 3rd International Software Metrics Symposium IEEE. 1996:90-99.
95. Briand LC, Daly J, Porter V, Wust J. A comprehensive empirical validation of design measures for object-oriented systems. Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262) IEEE. 1998:246-257.
96. Briand LC, Daly JW, Wust JK. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans Softw Eng*. 1999;25(1):91-121.
97. Briand LC, Wüst J, Daly JW, Porter DV. Exploring the relationships between design measures and software quality in object-oriented systems. *J Syst Softw*. 2000;51(3):245-273.
98. Alenezi M, Zarour M. Modularity measurement and evolution in object-oriented open-source projects. 2015.

AUTHOR BIOGRAPHIES



Bruno L. Sousa. He is a PhD candidate in Computer Science at Federal University of Minas Gerais. He holds a Master's degree in Computer Science at Federal University of Minas Gerais (2017) and a Bachelor's in Computer Science at Federal Institute of Science, Technology, and Education Southeast of Minas Gerais (2016). He has experience in Software Engineering, and his research focuses on software evolution, quality, metrics, maintenance, analytics, and empirical software engineering.

Mariza A. S. Bigonha. She is a Full Professor of the Computer Science Department at the Federal University of Minas Gerais. She holds a Bachelor's degree in Law (1975), a Master's degree in Computer Science from the Federal University of Minas Gerais (1985), and a PhD in Computer Science from the Pontifical Catholic University of Rio de Janeiro (1994). Her research concentrates on Programming Language and Software Engineering in the following research lines: programming language definition and implementation, compilation techniques, code optimization, software quality, software metrics, and software maintenance, software evolution, software analytics, and empirical software engineering.

Kecia A. M. Ferreira. She is a Professor of the Department of Computing at the Federal Center for Technological Education of Minas Gerais. She holds a PhD in Computer Science from the Federal University of Minas Gerais (2011), a Master's degree in Computer Science from the Federal University of Minas Gerais (2006), and a Bachelor's degree in Computer Science from the Pontifical Catholic University of Minas Gerais (1999). She has professional and academic experience in Software Engineering in the following research lines: software quality, software metrics, software maintenance, software evolution, software analytics, and empirical software engineering.

Glaura C. Franco. She is a Full Professor of the Department of Statistics at the Federal University of Minas Gerais. She holds a PhD in Electrical Engineering (Systems Theory–Time Series) from the Pontifical Catholic University of Rio de Janeiro (1998), a Master's degree in Applied Mathematics (Statistics) at IMPA (1990), and a Bachelor's degree in Statistics at the Federal University of Minas Gerais (1985). She has professional and academic experience in Statistics. She has research activities in time series, working mainly on the following topics: bootstrap, long dependence, and structural models.

How to cite this article: Sousa BL, Bigonha MAS, Ferreira KAM, Franco GC. Evolution of internal dimensions in object-oriented software–A time series based approach. *Softw: Pract Exper*. 2024;54(6):1034-1073. doi: 10.1002/spe.3310